Cromemco™ C

REFERENCE MANUAL

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View,CA 94043

Part No. 023-4029

February 1981

This manual was produced on a Cromemco System Three computer utilizing a Cromemco HDD-22 Hard Disk Storage System running under the Cromemco Cromix Operating System. The text was edited with the Cromemco Cromix Screen Editor. The edited text was formatted using the Cromemco Word Processing System Formatter II. Final camera-ready copy was printed on a Cromemco 3355A printer.

CROMEMCO C

Reference Manual

Table of Contents

INTRODUCTION

Cromemco C is a powerful general purpose programming language which has been written to conform as closely as practical to the C language described in the text **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie. Throughout this manual, the terms **reference text** and **reference language** are used to refer to that book and the C language described therein. Cromemco C has been developed using the C Reference Manual portion (Appendix A) of the reference text as the authoritative definition of the language. The specification in the Reference Manual is the final word. Any ambiguities in the specification itself have been resolved by reference to the User's Manual (chapters 0 - 7 of the reference text).

The purpose of this manual is to introduce the features and use of Cromemco C, and the differences between it and the reference language. **The C Programming Language** is the most complete source of information about C, and includes numerous examples along with the description of C. Those unfamiliar with C are referred to that book, a copy of which has been included as part of the Cromemco C package.

Many useful programs can be written without detailed knowledge of the operating system environment in which they will run, but many others need to utilize all facilities of the target operating system. Authors of the latter type of program are referred to the Cromemco Cromix Operating System Instruction Manual, part number 023-4022, and the Cromemco CDOS Instruction Manual, part number 023-0036, for descriptions of such system-dependent details as file name formats, system calls, and mass storage organization.

The Cromemco C compiler consists of three passes which translate a C source program into an assembly language source program. The assembly language program is then input to the Cromemco Macro Assembler which translates it to a relocatable object file. Finally, this relocatable file is loaded and linked with the Cromemco Link program to produce an executable object program.

The compiler must be run under the Cromix operating system, which requires the following minimum hardware configuration:

Cromemco computer system,
128K of memory,
CRT terminal,
a minimum of 243K of disk space.

A printer is optional. Please note that the compiler itself is too large to be executed under CDOS. However, it is possible to write and compile C programs which can then be linked to execute under CDOS.

A C program which has been linked using the CDOS C function library can be executed under CDOS or under the CDOS simulator program on the Cromix operating system. A C program which has been linked using the Cromix C function library will execute only on a Cromix system.

Chapter 1

C Synopsis

This chapter presents descriptions of those features available in Cromemco C. Features which are system or implementation-dependent are fully described; features which are already carefully described in the reference text are just briefly described. Chapter 3, Differences and Limitations, lists all aspects of Cromemco C which are in some way different from the reference language.

1.1 Names

static and **auto** names have 8 significant characters. The compiler ignores all characters beyond the eighth. Upper and lower case letters are distinct.

Macro names (the parameters of **#define**) are significant to all characters. Again, upper and lower case are distinct.

extern (global) names, including all function names, have only 7 significant characters. C recognizes the difference between upper and lower case, but the **asmb** and Link programs do not, so care must be taken to spell an extern name the same way, including case, everywhere it is used.

Names 1 or 2 characters long are prefixed with "$$$" when they are inserted into the Z80 file output from cp2 to ensure lack of conflict with Z80 register names.

1.2 Constants

Constants are formed as described in the reference text. Briefly:

char	a character enclosed in single quotes. Non-graphic characters can be represented using the backslash, \, and 1 to 3 octal digits, as described in the reference text, or backslash followed by 'x' and two hexadecimal digits.

<u>examples:</u>

```
'a'
'\n'
'\012'
'\x0a'
```

float an integer part, a decimal point, a fraction part, **e** or **E**, and an exponent.

<u>examples:</u>

```
5.
5e0
5E0
```

hexadecimal

one to eight hexadecimal digits preceded by the digits **0x** or **0X**.

<u>examples:</u>

```
0x1a
0Xffff
```

int one or more digits representing a value in the range -32768 to 32767.

<u>examples:</u>

```
100
-5
```

long an integer, octal, or hexadecimal constant followed by **L** or **l**, or an integer, octal, or hexadecimal constant too large to be represented as an **int**.

<u>examples:</u>

```
80000
70000L
0Xa5a5a5a5L
```

octal an integer constant preceded by the digit 0.

<u>examples:</u>

```
0177
0912
```

string one or more characters enclosed within double quotes. C adds a null byte after the final character.

<u>examples:</u>

```
"a"
"the value is %d\n"
```

1.3 Storage Class Specifiers

The available storage class specifiers are

auto

extern

register

static

typedef

register is allowed by the compiler, but it is always functionally identical to **auto.**

The default storage class is **auto** inside a function, including the **main** function, and **extern** outside, except that functions are never **auto.** Note that a default **extern** declaration does not have quite the same effect as an explicit **extern** declaration: explicitly declaring a variable **extern** does not <u>define</u> it, i. e., allocate storage for it. An external variable is defined by declaring it outside of any function, either in the same module or in a different module, without the **extern** specifier.

extern variables and all functions are accessible to all statements in all modules which comprise a program, whereas a variable defined as **static** outside of any function within a module is accessible only to statements within that one module.

1.4 Type Specifiers

The available type specifiers are:

char
short
int
unsigned
long
float
double
structure specifier
union specifier

typedef name.

The various types of data have the following characteristics in Cromemco C:

char	one byte (8 bits) long. The most significant bit is not propagated when a **char** is converted to an **int.** The range of values of a **char** is 0 to 255.
short	the same as **int.**
int	two bytes (16 bits) long, stored LSB, MSB in the Z80 convention. The range of values of an **int** is -32768 to 32767.
unsigned	the same as **int** except that the range of values is 0 to 65535.
long	four bytes (32 bits) long, stored LSB,...,MSB. All **longs** are assumed to be signed. The range of values is approximately -2E9 to 2E9.
float	four bytes (32 bits) long, with 6 digits of accuracy and values in the range +/- 9.99E-65 to +/- 9.99E+62. The four bytes are stored in the order: sign and base-10 exponent, most significant byte of mantissa, second byte of mantissa, least significant byte of mantissa. The most significant bit in the first byte stores the sign; 0 indicates a positive number, 1 indicates a negative number. The exponent is stored in excess-64 (excess-40H) format (64 decimal is added to the real exponent to form the stored exponent). The next 3 bytes contain the BCD (Binary Coded Decimal) mantissa, 2 digits to the byte, normalized to a value between .1 and 1. The implicit decimal point is located before the first digit of the mantissa.
double	eight bytes (64 bits) long with 14 digits of accuracy and values in the range +/- 9.99E-65 to +/- 9.99E+62. The first byte contains the sign and exponent as with **float,** and the next 7 bytes contain the 14 digit BCD mantissa, most significant byte first.

1.5 Declarations

Declarations can be simple or arbitrarily complicated, as described in the reference text. Briefly, a declaration consists of an optional storage class specifier, one or two type specifiers, and a list of declarators matched, optionally, with initializers.

There are just four cases in which two type specifiers are legal:

short int
long int
unsigned int
long float

A declarator is defined recursively to consist of one of the following:

an identifier (name),

a declarator enclosed within parentheses to override the default binding of operators used within a declarator,

a declarator preceded by an asterisk, *****, which indicates that the declarator is a pointer,

a declarator followed by parentheses, which indicates that the declarator is function,

or a declarator followed by square brackets, which indicates the declarator is an array. The square brackets contain an optional constant expression which declares the size of the dimension, which can be a maximum of 32,767 (the maximum value of an integer).

<u>examples:</u>

```
     int i;
```

declares i to be an integer. The storage class would be **auto** within a function, and **extern** outside. This declaration allocates storage whether inside or outside of a function.

```
     extern int i;
```

declares that i is an integer and that it is defined outside of the function or even in another module. No storage is allocated for an item explicitly declared **extern.**

```
int *pi;
```

declares pi to be a pointer to an integer value.

```
int fi ();
```

declares fi to be a function which returns an integer value.

```
int *fpi ();
```

this declares fpi to be a function which returns a pointer to an integer value.

```
int (*pfi) ();
```

this example shows the use of parentheses to override the default binding. pfi is declared to be a pointer to a function which returns an integer value.

```
int *(*pfpi) ();
```

declares pfpi to be a pointer to a function which returns a pointer to an integer value.

```
int ai [10];
```

ai is an array of 10 integers.

```
int ai [10][5];
```

ai is a two-dimensional array; it contains 10 arrays, each of which contains 5 integers.

```
extern int ai [];
```

ai is declared to be an array of integer values. ai and its size are defined elsewhere, so it's not necessary to declare the size here.

```
int *api [10];
```

api is an array of pointers to integer values.

```
int (*pai) [10];
```

pai is a pointer to an array of 10 integers.

1.6 Structure and Union Declarations

Structure and union declarations, again, can be simple or arbitrarily complicated, as fully described in the reference text.

Briefly, a structure declaration consists of a storage class specifier, the keyword **struct**, an optional name, called the structure tag, which can be used in subsequent declarations of other structures, and one or more structure declarators separated by semicolon, **;**, and enclosed within braces, { }. A structure declarator consists of a type specifier and one or more structure members, separated by commas. A structure member is a declarator as defined in the previous section.

Structure and union specifiers are type specifiers, which means that structures can be declared which contain other structures and unions.

A union declaration has the same form as a structure definition, except that the keyword **union** is used.

examples:

```
     struct stag {
          int i,
              j;
          char c;
     } s;
```

this declares s to be a structure consisting of two integers and a character. stag is the structure tag.

```
     struct stag s2;
```

this declares s2 to be a structure having the same form as s.

```
     struct atag {
          char c;
          float f;
     } ;
```

this declares a structure tag, atag, for future use.

```
     struct {
          char *pc;
          long l [5][5];
          long (*pfl) ();
     } sname;
```

as seen here, a structure member can be arbitrarily complicated. This structure contains a pointer, an array, and a pointer to a function.

```
     struct {
          int flag;
          struct {
               double d;
               char c;
          } s2;
     } sl;
```

sl is a structure which contains an integer and another structure, s2.

```
     struct {
          char a;
          long l;
     } *ps;
```

ps is declared to be a pointer to a structure with the specified form.

```
     struct {
          char a;
          long l;
     } as [5];
```

this declares as to be an array containing five structures of the specified form.

```
     struct {
          char a;
          long l;
     } *aps [5];
```

this declares aps to be an array of five pointers to structures of the specified form.

```
     struct {
          char a;
          long l;
     } *f ();
```

this declares f to be a function which returns a pointer to a structure of the specified form.

1.7 Initializers

Cromemco C implements intialization as described in the reference text, with the enhancement that **auto** aggregates (arrays and structures) can be initialized.

Briefly, an initializer consists of an equal sign, =, followed by a set of values. The set of values can be one expression or a list of initializers separated by commas and enclosed within braces, { }. An individual initializer can be an expression, or another list of initializers, also enclosed within braces.

examples:

```
     float f = 17.37E10;

     char c = 'r';

     char d = 'X';

     char s[] = "a string";

     char *ps = "a string";

     int ai [3] = { 1, 2, 3 };

     int ai [3][3] = {
          { 1, 2, 3 },
          { 4, 5, 6 },
     }
```

this initializes the first two rows of ai; the final row will be initialized to zeroes.

```
     struct {
          char c;
          int i;
          long l;
     } s = { 'a', 941, 100000L };
```

This shows the initialization of the members of a structure.

1.8 Operators

Cromemco C offers the following operators. Detailed descriptions and examples of the use of these operators can be found in the reference text.

Assignment operators:

An assignment statement of the form

expl op= exp2

is equivalent to the statement

expl = expl op exp2

except that **expl** is evaluated only once.

=	simple assignment;
+=	addition;
-=	subtraction;
*=	multiplication;
/=	division;
%=	remainder division;
>>=	right shift;
<<=	left shift;
&=	bitwise **and**;
^=	bitwise exclusive **or**;
|=	bitwise inclusive **or**;

Additive operators:

+	addition;
-	subtraction;

Bitwise operators:

&	and;
^	exclusive or;
|	inclusive or;

Comma operator:

exp, exp	evaluate and discard the left expression, then evaluate the right expression;

Conditional expression operator:

exp ? expl : exp2

the value of a conditional expression is **expl** if **exp** is true, else **exp2**;

Logical operators:

&&	and;
||	or;

Multiplicative operators:

*	multiplication;
/	division; integer division truncates towards 0 in all cases;
%	remainder division;

Relational operators:

<	less than;
<=	less than or equal to;
>	greater than;
>=	greater than or equal to;
==	equal to;
!=	not equal to;

Shift operators:

<<	shift left;
>>	shift right; right shift of signed item fills vacated bits with sign bit

Unary operators:

*	indirection;
&	address pointer;
-	arithmetic negation;
!	logical negation; produces **int** 0 or 1;
~	one's complement;
++	pre and post-increment;
--	pre and post-decrement;
(typename)	cast;
sizeof	size in bytes;
sizeof (typename)	

1.9 Control Flow

The semicolon, **;**, terminates each statement.

Braces, { }, are used to group statements and declarations into a single compound statement, also called a block. The semicolon is not required following a compound statement.

Recursive function calls are supported.

The following statements are available in Cromemco C.

```
; (the null statement)

break;

continue;

do statement while (expression);

for (expression1; expression2; expression3)
     statement

goto label;

if (expression)
     statement1
else
     statement2

return;
or
return expression;

switch (int-expression) {
     case constant-expression:  statement
     .
     .
     default:  statement
}

while (expression)
     statement

identifier :  (labeled statement)
```

1.10 #define and #undef

#define allows the programmer to specify text which will replace the defined text and to define macros with 0 or more arguments. **#define** has all the flexibility and pitfalls as described in the reference text, with the limitation that a **#define** line, with continuation lines, can contain a maximum of 512 characters. Use \ to continue long **#defines**. cp0, the preprocessor pass of the compiler, will discard the \ and continue with the first character on the next line.

#undef identifier causes cp0 to forget the definition of **identifier**.

1.11 #ifdef, #else, #endif

#ifdef, #else, and **#endif** have the same function as described in the reference text, that of conditional inclusion and exclusion of C source lines. **#if** is treated as **#ifdef** and will draw a warning message whenever it is used.

1.12 #include "filename" #include <filename>

As in the reference language, **#include** directs the compiler to replace the **#include** line with the contents of **filename** and continue the compilation. The compiler will look for **filename** in the current directory unless the **-i** or **-l** options are entered to the first pass, cp0. **-i dirname** will cause cp0 to add **dirname** to the front of all included filenames enclosed within double quotes, "". **-l dirname** does the same for all included filenames enclosed within <>. The default directory indicated by "" is the current directory; the default indicated by <> is /usr/include/.

1.13 Command Line Arguments

C programs may use **argc** and **argv,** as described in the reference text, to obtain the values of the command line arguments. CDOS and the CDOS simulator do not pass the name of the program as the 0th argument, so **argv[0]** is not available to C programs linked to run under CDOS. CDOS and the CDOS simulator both convert all lower case characters in the command line to upper case. The Cromix operating system provides the command line arguments as described in the reference text.

Chapter 2

EXTENSIONS

This chapter describes the ways in which Cromemco C has been extended from the reference language.

2.1 #control

#control is a compiler directive which has the form

```
#control command line
```

where **command line** is either a valid compiler control command, the name of an assembly language macro, or a comment to be inserted into the assembly language source output from pass 2 of the compiler. The **#** must be in column 1 of the input line.

In the following list of valid commands, letters in upper case must be used in the command. Letters in lower case are optional. Actual commands may be given in either upper or lower case. **asmb** refers to the Cromemco Macro Assembler program.

command	recognized by pass	action
BAsm	0,2,asmb	start a block of embedded assembly language code. The compiler will pass the next block of source lines to the Z80 output file without compilation.
EAsm	0,2,asmb	end embedded assembly language block.
LIst	asmb	generate a "LIST ON" in the Z80 output file, which directs asmb to produce a listing. LIst remains in effect until the next NList.
NList	asmb	generate a "LIST OFF" in the Z80 output file, which directs asmb to suppress the listing. NList remains effective until the next LIst.
MAcro	asmb	generate a "LIST GEN" in the

		Z80 output file, which directs asmb to list the expansion of each assembly language macro following each macro call. MAcro remains effective until the next NMacro.
NMacro	asmb	generate a "LIST NOGEN" in the Z80 output file, which directs asmb not to list macro expansions. NMacro remains effective until the next MAcro.
SOurce	0,2,asmb	include C source in the Z80 output file as comments. Each C source line will precede the resulting assembly language code generated for the line.
NSource	0,2,asmb	do not include C source in Z80 output file.
B1	1	display C source line numbers on terminal as encountered. (Diagnostics aid.)
B2	1	stop displaying line numbers. (Diagnostics aid.)
B5	1	display file names on terminal as encountered. (Diagnostics aid.)
B6	1	stop displaying file names. (Diagnostics aid.)

The compiler assumes that a command line which begins with an underline character is the name of an assembly language macro and will insert the call unchanged in the Z80 output file for expansion by the assembler. No arguments can be passed to the macro with this method.

C will insert any other command line as a comment in the Z80 output file if source is included in the output file. The command line may contain spaces.

An embedded assembly language section should not contain any compiler directives, e.g., **#control**, **#define**, and **#ifdef**.

2.2 Initialization of Auto Aggregates

Cromemco C permits **auto** arrays and **auto** structures to be intialized. The initialization is done each time the function is entered. Initialization of **auto** variables is done at runtime by code generated by the compiler, which means that initialization of very large aggregates can be prohibitively costly in terms of generated code.

2.3 In-line Assembly Code

Some applications require code efficiency or capabilities not available in C; therefore, Cromemco C includes the capability of intermixing assembly language code with C code.

A block of assembly language code in a C program must be preceded by the line

```
#control basm
```

and followed by the line

```
#control easm
```

Embedded assembly code can use the value of a preceding C expression, which is always left in one or more Z80 registers:

a **char** value is left in L;

an **int** value is left in HL;

an **unsigned** value is left in HL;

a pointer value is left in HL;

a **long** value is left in DEHL (D is MSB, L is LSB);

a **float** value is left in the **extern** variable $FR0;

a **double** value is left in the **extern** variable $FR0;

Note that the IX register is used by the surrounding C function to access its **auto** variables and must therefore be preserved by the assembly language code.

Embedded assembly code may refer to the name of a variable defined in the surrounding C source only when that name is <u>defined</u> as external in the C source and declared **global** within the embedded assembly code.

Recall from Section 1.1 of this manual that C adds the prefix "$$$" to names one and two characters long, so embedded assembly code must use the augmented name.

example:

An argument may be passed to embedded assembly language code by using a C expression, the evaluation of which leaves the value of the expression on the TOS.

```
        unsigned port, data;      /*declarations */

        (port << 8) + data;       /*an expression*/

#control basm
        ld      c,h               ; port is in h (MSB of TOS)
        out     (c),l             ; data is in l (LSB of TOS)
#control easm
```

2.4 Structures

In a context requiring the address of a structure, the structure name by itself may be used as a pointer to the structure.

This can be convenient when using a pointer to a structure as a function argument.

example:

```
    struct
        {
        int i;
        int j;
        } structure;

    f(structure)
```

is equivalent to

```
    f(&structure.i)
```

Chapter 3

DIFFERENCES AND LIMITATIONS

This chapter lists those items available in the reference C language which are not available or which are different or limited in some way in Cromemco C.

3.1 #define

A **#define** line, with continuation lines, can contain a maximum of 512 characters.

3.2 #if

#if has been implemented as **#ifdef.** The compiler will issue a warning if **#if** is found in a program.

3.3 Bit Fields in Structures

Bit fields in structures are not available and will be treated as an error by the compiler if they appear in a program.

3.4 Casts

The use of casts (e.g., **(long) i**) is limited to the fundamental data types **char, short, int, unsigned, long, float,** and **double.**

3.5 Function Arguments

The maximum number of function arguments is restricted to 32.

3.6 Initializers

External structures and arrays may not contain initializers which are references to the containing structure or array.

3.7 String Length

The maximum string length is 255 characters, not including the null byte added at the end by the compiler.

3.8 Structures

A structure element name must be uniquely associated with the structure or structure tag within which it is defined.

example:

```
    struct { int I; ...} stl;

    struct { long L; ...} st2;
```

A reference to **stl.L** is invalid, and the compiler will treat it as an error.

3.9 switch

switch has a fixed maximum number of cases. The maximum is 32. Nested **switch** statements may each have up to 32 cases.

Chapter 4

INPUT AND OUTPUT

This chapter describes all of the I/O functions available in the Cromemco C function library as well as those details of input and output which one must know in order to write programs which read, write, and store data. As will be seen in this chapter and in Chapter 6, there are many possible ways to perform I/O in a C program. A program can access a single character, a line of characters, or an arbitrarily large block of characters using buffered I/O, unbuffered I/O, or Cromix system calls. This rich selection might confuse a programmer new to C, so here are some general guidelines for the use of I/O functions.

I/O functions fall into three basic categories: buffered, unbuffered, and Cromix system call interface functions. The buffered functions (e.g. **printf, scanf, putc, getc**) are the most commonly used of the three types, as illustrated by the programs in the reference text. There are buffered functions which access one character at a time (**getc, putc**), one line at a time (**fgets, fputs**), and perform formatted input and output (**printf, fprintf, scanf, fscanf**). There are, however, no buffered functions which can read or write an arbitrarily large block of characters at one time. Another disadvantage is that a program cannot open a file for read and write (also known as update) access.

There are unbuffered functions which access one line (**getl, putl**) or a block (**read, write**) at one time. There are no single-character unbuffered functions, so a C program which needs both to read and write large blocks of data and to use one character at a time must contain routines to unbuffer and buffer single characters.

The Cromix call interface functions have the least overhead of the functions in the three categories. There are functions which read and write a single character (**rdbyte, wrbyte**), one line (**rdline, wrline**), and a block of characters (**rdseq, wrseq**). There are two disadvantages to the use of these functions: a program which uses them cannot be compiled to run under CDOS, and there are no functions which perform formatted I/O.

In most cases, a C program cannot successfully mix buffered functions with unbuffered functions or Cromix calls to access one file. One can mix them successfully only when writing to **stdout** or **stderr.**

4.1 I/O Header Files

A C program which uses functions from the C library must include one of the header files **stdio.h** or **cdstdio.h**. **stdio.h** is used when compiling a program to run on a Cromix system, and **cdstdio.h** is used when compiling a program to run on a CDOS system. The two files differ mainly in their declarations of **FILE** and **File,** specially defined types which are discussed in the next section. Except as noted below, any C program which runs successfully on a Cromix system can be changed to run on a CDOS system simply by including the **cdstdio.h** header file instead of **stdio.h,** recompiling, and linking with the CDOS C function library. The program will run the same under CDOS so long as all declarations of file pointers made to use the buffered I/O functions use the **FILE** and **File** types defined in the header files.

The Cromix C function library contains interface functions which allow a C program to perform Cromix system calls directly. The CDOS function library does not, so a C program which uses Cromix system calls cannot simply be recompiled and relinked with the CDOS C function library to get a program which will run the same way on a CDOS system.

See Chapter 7, Standard Header Files, for listings of **stdio.h** and **cdstdio.h.**

4.2 File Numbers, File Pointers, and FILE Structures

The communication between a C program and a C I/O function as to which file to use is accomplished by use of either a file number or a pointer to a special structure called a **FILE** structure. A file number is an integer, declared by

```
    int filenumber;
```

and a **FILE** pointer is declared by

```
    FILE *filepointer;
```

One can also use the type **File,** defined in the **stdio.h** header file, to declare the pointer:

```
    File filepointer;
```

The **FILE** and **File** types are declared in both of the header files **stdio.h** and **cdstdio.h.**

Functions in the Cromix C function library actually use the **FILE** structure, whereas the functions in the CDOS C function library do not use it. All I/O functions in the CDOS C library use file numbers. The two header files contain different definitions of **FILE** and **File,** which makes it easy to change a program which runs under one of the Cromemco operating systems to run under the other. The C program is simply changed to include the appropriate header file, recompiled, and linked with the other C function library. This will yield a correct program so long as all declarations of file pointers were made using the **File** type from the header file.

A C program which runs on a Cromix system will run the same on a CDOS system unless it uses special knowledge about the **FILE** structure and performs some chicanery with the value of a member of the structure. This will not work when the program is compiled and linked to run under CDOS.

File pointer functions which access the **stdin, stdout,** and **stderr** files must use the **stdin, stdout,** and **stderr** file pointers as declared in the **stdio.h (cdstdio.h)** header file. File number functions must use the numbers **STDIN, STDOUT,** and **STDERR,** also defined in the header files.

Note that C file numbers are the same thing as Cromix channel numbers.

Some functions use or return file numbers, others expect or return a pointer to a **FILE** structure.

<u>functions which use file numbers</u>

creat
open
close
read
write
lseek

getl
putl

<u>functions which use **FILE** pointers</u>

fopen
fclose
getc
putc
ungetc
fgets
fputs
fprintf
fscanf
fseek

fillbuff
flushbuff

4.3 Buffered I/O versus Unbuffered I/O

In a C program compiled and linked to run on a Cromix system functions which use **FILE** pointers perform buffered I/O. This means that characters are saved up in an intermediate buffer before being made available to a C program on input or before being delivered to the operating system on output. Each file has a buffer allocated to it when the file is opened. The buffer will be filled or emptied by the functions as necessary, although a C program can call **flushbuff** to force a partial buffer to be written.

The C library arranges for the allocation of up to 20 file buffers, which means that a maximum of 20 buffered files, including **stdin**, **stdout**, and **stderr**, may be open at any one time. The three standard I/O files are always open, which means that a C program can explicitly open only 17 files.

In a C program compiled and linked to run on a Cromix system, functions which use file numbers perform unbuffered I/O, which means that the characters are delivered immediately to the C program on input or the operating system on output. There are no intermediate buffers, which means that the there may be as many files opened at one time as allowed by the Cromix system.

Unlike the Cromix operating system, CDOS requires all programs to supply their own file buffers, so one file management mechanism is used by all I/O functions in the the CDOS C library. This library arranges for up to 10 file buffers, which means that a maximum of 10 files may be open at one time. This maximum can be changed by including the following external definition in one of the C source modules, outside of any function:

```
int _maxfnum = n;
```

This definition of **_maxfnum** allows up to **n** + 1 files to be open at one time. Each buffer requires about 200 bytes of memory when the program executes, whether or not the buffer is ever used by a file. Since the CDOS C library also contains a definition for **_maxfnum,** the C module containing the new definition must be linked with the rest of the program before the CDOS C library is searched.

There is no guarantee that a C program will yield predictable results if it performs unbuffered and buffered I/O simultaneously on the same file.

stdin is defined (in **stdio.h**) to be buffered, so that redirected input will execute at high speed. **stdout** and **stderr** are defined to be unbuffered.

4.4 Filenames

Filenames used with Cromemco C and in C programs are either CDOS file names or Cromix file names. Refer to the appropriate manual for details. See the Introduction to this manual for the manual part numbers. The CDOS simulator, which executes CDOS programs under the Cromix operating system, converts all disk drive identifiers to Cromix directory names according to the following table:

drive identifier	directory
A:	current directory
B:	/b/
C:	/c/
D:	/d/
E:	/e/
F:	/f/
G:	/g/
H:	/h/

4.5 stdin, stdout, stderr

stdin, stdout, and **stderr** are three files always available to each C program for standard input, standard output, and standard error output. They are opened automatically when the program begins execution. A C program on a Cromix system which uses buffered I/O functions to access **stdin, stdout,** and **stderr** must use their correct **FILE** pointers, which are defined in **stdio.h.** A program which uses unbuffered functions must use the file numbers for **stdin, stdout,** and **stderr.**

These numbers are defined in **stdio.h** and **cdstdio.h** as **STDIN, STDOUT,** and **STDERR.**

These files are by default connected to the user's terminal, although they can all be redirected using the redirected I/O facility. When using redirection under the Cromix operating system, the user can choose to have **stderr** redirected with **stdout,** or he can leave **stderr** connected to the terminal. When running under CDOS or the CDOS simulator, **stderr** is always redirected with **stdout.** CDOS itself does not support redirected I/O--redirection under CDOS and the CDOS simulator is supported by the CDOS C function library.

When a program is executed as a detached process in a Cromix system, **stdin** is connected to the null file, which always returns EOF, if no input redirection has been done.

4.6 Device Names

A Cromemco C program which is linked to execute under CDOS or the CDOS simulator can access the system line printer and the system console by treating them as files with special names. The line printer has the name $LP and the console has the name $CON.

A C program can write to the line printer by using **open** or **fopen** to open the file named $LP for write access and obtaining a file number or file pointer. This file number or pointer is then used in calls to **fprintf, write,** and **putc.**

A C program can write to the console in a similar way by using the file named $CON, although **printf** is the first choice for such output.

4.7 Carriage Return, Newline, and EOF

Many files used in Cromemco systems, including text files, contain records each of which is terminated by a carriage return, line feed pair. Existing programs which expect the data bytes to be followed by just a newline (line feed character) must be changed to discard any preceding carriage return. The **getl** function in the Cromix C library automatically converts a carriage return, newline pair to a single newline. Programs which will be linked to run under CDOS must also expect from 0 to 127 CONTROL-Z characters (0x1A) after the final carriage return, line feed pair before EOF is detected.

4.8 I/O Functions

The following functions are available in both the CDOS C function library and the Cromix C function library. These libraries, named CDOSCLIB.REL and CLIB.REL, respectively, are supplied with the C package. A C program may use any of these functions just as it may use any other externally defined function.

A C program which refers to a standard I/O function such as **printf** must **#include** either **stdio.h** or **cdstdio.h**, depending on whether it will compiled to run on a Cromix system or a CDOS system. Listings of **stdio.h** and **cdstdio.h** appear in Chapter 7, Standard Header Files.

The following function descriptions contain first the function name and argument list, the types of the arguments, a description of the function's behavior, and a list of possible return values.

backc(fp, c)

```
File fp;
char c;
```

backc is the same as **ungetc,** which pushes one character back onto a file's buffer. Just one character can be pushed, and no error is reported if more than one is pushed--previously pushed characters are lost.

Returns **char:**

c.

close(fn)

```
int fn;
```

close closes an open file and is implemented as described in the reference text. **fn** is the file number.

Returns **int:**

0 if no error occurred;
-1 if the file was not open.

creat("name", pmode)
or
creat(fname, pmode)

```
**char** *fname;
**unsigned** pmode;
```

creat creates a file with the specified name and opens it for write access only. An old file with the same name is deleted.

creat is implemented as described in the reference text, with the exception that **pmode,** the protection mode specifier, can specify either CDOS file protection or Cromix file protection.

pmode is an unsigned variable or constant which specifies the settings of the file's access privilege bits. The bits of the value have different meanings, depending on whether the program will be linked to execute under CDOS or the Cromix operating system. A **pmode** of 0 instructs **creat** to let the operating system assign the default values to the access bits.

For the Cromix operating system, the bits have the following meaning:

```
bits 15 through 12 are 0

bit 11 - owner append access
bit 10 - owner write access
bit 9  - owner execute access
bit 8  - owner read access

bit 7  - group append access
bit 6  - group write access
bit 5  - group execute access
bit 4  - group read access

bit 3  - other append access
bit 2  - other write access
bit 1  - other execute access
bit 0  - other read access
```

Bit 15 is the most significant bit.

For CDOS, the bits have the following meaning:

```
bits 15 through 8 are 0

bit 7  - erase protect
bit 6  - write protect
bit 5  - read protect
```

bit 4 - user file

bit 3 is system file
bits 2 through 0 are 0.

Returns **int**:

file number of successfully created file;
-1 if an error occurred.

fclose(fp)

```
File fp;
```

fclose closes an open file. **fp** is the **FILE** pointer. Any buffered characters are flushed to disk before the file is closed.

fclose is declared in the **stdio.h** header file as

```
File fclose();
```

Returns:

fp if no error occurred;
NULL if the file was not open.

fgets(line, maxbytes, fp)

```
char *line;
int maxbytes;
File fp;
```

fgets reads the next line (characters followed by \n), including the \n, from the file with **FILE** pointer **fp** and puts it into **line** followed by \0. **fgets** will read a maximum of **maxbytes** - 1 characters.

fgets is declared in the **stdio.h** header file as:

```
char *fgets();
```

Returns pointer to **char**:

line (the original pointer) if no error occurred;
NULL if EOF and no bytes read.

fillbuff(fp)

```
File fp;
```

fillbuff causes C to fill the buffer of the file indicated by **fp** using bytes starting at the current file location. Only specialized applications need to use this function, and care must be taken not to fill the buffer if it is not already empty. C automatically fills file buffers when necessary.

Returns **int:**

number of bytes read;
-1 if an error occurred.

flushbuff(fp)

```
File fp;
```

flushbuff writes all bytes from the buffer of the file indicated by **fp** out to disk. As with **fillbuff,** only specialized applications need to use this function. C automatically flushes file buffers when necessary.

Returns **int:**

number of bytes written if no error;
-1 if an error occurred.

fopen("name", "mode", blocksize)
or
fopen(fname, openmode, blocksize)

```
char *fname;
char *openmode;
unsigned blocksize; (optional)
```

mode has the form:

```
[x][rwa][b][e]
```

where

x = exclusive access (optional);

r = read;
w = write;
a = append;

b = blocked file (optional);

e = abort on I/O error (optional).

fopen opens the file named **name** for buffered I/O with read, write, or append access. Combinations of read, write, and append are not allowed. The access is non-exclusive unless **x** is used in the mode string. Opening a file with write access will delete an existing file with the same name or create the file if it doesn't already exist. Opening a file with append access will create the file if it doesn't already exist.

Exclusive access is available only to programs which run on a Cromix system--CDOS doesn't have such a feature. The "xr" access mode causes the file to be opened with full exclusivity--no other program will be allowed to read, write, or append to the file. The "xw" and "xa" modes cause the file to be opened with write and append exclusivity--no other program will be allowed to write or append to the file.

A maximum of 20 buffered files may be open at any one time, including the three files **stdin, stdout,** and **stderr,** which are always open.

The **b** mode specifier is used to indicate that reads and writes to the file will be buffered in blocks which are **blocksize** bytes long. When **b** is used, the optional argument **blocksize** must be used also. C uses a blocksize of 512 when **b** is not specified, which is usually the most convenient size, although some applications might specify a larger block in order to achieve slightly higher throughput. The maximum value of **blocksize** is 32,767.

The **e** mode specifier instructs C to abort with an error message should it encounter an error while reading or writing bytes onto the file using any of the I/O functions which use a **FILE** pointer. By default, C returns an error value to the program after an I/O error occurs.

The append mode, **a,** is not available to programs which will be linked to run under CDOS.

fopen is declared in the **stdio.h** header file as:

```
File fopen();
```

Returns pointer to **FILE**:

FILE pointer of the successfully opened file;
NULL if the file could not be opened.

fprintf(fp, cstring, arg1, arg2, ...)

```
File fp;
char *cstring;
```

fprintf writes formatted output to the file with **FILE** pointer **fp** and is implemented as described in the reference text, with the same extensions as printf.

Returns **int**:

-1 if an error occurred.

fputs(line, fp)

```
char *line;
File fp;
```

fputs writes an output string, which must be terminated by \0, from **line** to the file with **FILE** pointer **fp.** The \0 is not written.

fputs is declared in the **stdio.h** header file as:

```
char *fputs();
```

Returns pointer to **char**:

line if no error;
NULL if an error occurred.

fscanf(fp, cstring, arg1, arg2, ...)

```
File fp;
char *cstring;
```

fscanf reads characters from the file with **FILE** pointer **fp,** converts them according to the conversion specifications in **cstring,** and stores them into the locations pointed to by the remaining arguments, which must be pointers. **fscanf** is implemented as described in the reference text.

Returns **int:**

number of fields scanned if no error;
-1 if an error occurred.

fseek(fp, offset, origin)

File fp;
long offset;
int origin;

Origin specifier values are:

0 = from the beginning;
1 = from the current location;
2 = from the end.

fseek sets the file position pointer (which points to the next character position to be read or written) to **offset.** **fseek** is the same as **lseek** except that it is used to set the file position of a file being accessed with the buffered I/O functions (**fopen, getc,** and so on).

origin type 2 is not available to programs which will be linked to run under CDOS. **fp** is the file pointer.

fseek is declared in the **stdio.h** header file as:

long fseek;

Returns **long:**

current location if no error;
-1 if error or attempt to seek back from beginning.

getc(fp)

File fp;

getc reads the next byte from the file with **FILE** pointer **fp. getc** returns a binary value with type **int:** the 8th bit is not stripped from the input byte and control characters have no special meaning. **getc** does not strip the carriage return which precedes the line feed (newline) at the ends of records in many files on Cromemco systems. See 4.7, Carriage Return, Newline, and EOF, for a discussion of EOF detection.

Returns **int**:

next character from **fp** if no error;
-1 if an error or EOF occurred.

getchar( )

getchar is defined to be **getc(stdin)**.

Returns **int**:

next character from **stdin** if no error;
-1 if an error or EOF occurred.

getl(fn, buf, maxbytes)

```
int fn;
char *buf;
int maxbytes;
```

getl reads the next input line, including the newline character, from the file with number **fn** into the character array **buf**. The line in **buf** will be terminated with 0, and no more than **maxbytes** - 1 characters will be read. **getl** discards a carriage return which immediately precedes the newline.

getl returns the number of characters read, unlike **fgets**.

Returns **int**:

number of characters read, including the \0;
0 if any error or EOF occurred.

getline(buf, max)

```
char *buf;
int max;
```

getline is **#define**d to be **getl(stdin, buf, max)**.

Returns **int**:

number of characters read, including the \0;
0 if any error or EOF occurred.

lseek(fn, offset, origin)

```
int fn;
long offset;
int origin;
```

Origin specifier values are:

```
0 = from the beginning;
1 = from the current location;
2 = from the end.
```

lseek sets the file position pointer (which points to the next character position to be read or written) to **offset.** **lseek** is implemented as described in the reference text, with the enhancement that it returns the current offset as type **long.** This offers an easy way to determine the current file position:

```
lseek( fn, 0L, 1 )
```

lseek is the same as **fseek** except that it is used to set the file position of a file being accessed with the unbuffered I/O functions (**open, read,** and so on).

origin type 2 is not available to programs which will be linked to run under CDOS. **fn** is the file number.

lseek is declared in the **stdio.h** header file as:

```
long lseek;
```

Returns **long:**

```
current location if no error;
-1 if error or attempt to seek back from beginning.
```

open("name", rwmode)
or
open(fname, rwmode)

```
char *fname;
int rwmode;
```

Open mode values:

```
0 = read;
1 = write;
2 = read and write.
```

open opens an unbuffered file for read, read and write, or write access and is implemented as described in the reference text. The number of open files is limited by the operating system, not by some characteristic of the C library as it is with buffered files.

Returns **int**:

file number of successfully opened file;
-1 if an error occurred.

printf(cstring, arg1, arg2, ...)

char *cstring;

printf writes formatted output to **stdout**. It provides the features of the **printf** described in the reference text, along with the following extensions:

There is an additional conversion specification, **%b**, which takes 2 arguments: a string pointer and a string length. **%b** writes only as many characters from the string as specified by the length argument.

%l by itself is a legal conversion specification--it has the same meaning as **%ld. %ld, %lx, %lo**, and **%lu** all work as defined in the reference text.

There is an additional justification character, **+**, which causes the converted argument to be right justified. + is equivalent to having no justify character.

Any character with a value less than or equal to '0', with the exception of '.', can be used to pad a field not filled by the converted argument. For example, **%!10d** will cause the use of ! as the fill character.

The **%e, %f,** and **%g** conversion specifications produce slightly different output than the **printf** described in the reference text:

The **%f** specification will produce at most 16 digits to the left of the decimal point and 14 digits to the right of the decimal. A number which would be converted to have more than 16 digits will be converted using the **%e** specification. A precision greater than 14 is treated as 14.

The **%g** specification will not suppress trailing zeroes, and the **%e** specification will be used for all numbers less than 1.E-2 or greater than or equal to 1.E6.

Returns **int:**

-1 if an error occurred.

putc(c, fp)

```
char c;
File fp;
```

putc writes one character into the next location on the file with **FILE** pointer **fp** and is implemented as described in the reference text.

Returns **int:**

-1 if an error occurred.

putchar(c)

```
char c;
```

putchar is defined to be **putc(stdout)**.

Returns **int:**

-1 if an error occurred.

putl(fn, buf)

```
int fn;
char *buf;
```

putl writes the line in **buf** onto the file with file number **fn.** The line must be terminated by \n or \0. The terminator will be written to the file.

Returns **int:**

number of bytes written;
-1 if any error occurred.

read(fn, buf, n)

```
int fn;
char *buf;
int n;
```

read reads **n** bytes from the file with file number **fn** into **buf.**

Returns **int:**

number of bytes read if no error;
0 if EOF;
-1 if an error occurred.

scanf(cstring, arg1, arg2, ...)

char *cstring;

scanf reads characters from **stdin** and interprets them according to the conversion specifications in the control string. The converted values are stored in the locations pointed to by the remaining arguments, which must be pointers. **scanf** provides the features of the **scanf** described in the reference text.

Returns **int:**

number of fields scanned if no error;
-1 if an error occurred.

sprintf(string, cstring, arg1, arg2,...)

char *string;
char *cstring;

sprintf writes formatted output into **string** and is implemented as described in the reference text, with the same extensions as printf.

Returns **int:**

-1 if an error occurred.

sscanf(string, cstring, arg1, arg2,...)

char *string;
char *cstring;

sscanf reads characters from **string,** converts them according to the conversion specifications in **cstring,** and stores them into the locations pointed to by the remaining arguments, which must be pointers. **sscanf** is implemented as described in the reference text.

Returns **int:**

number of fields scanned if no error;
-1 if an error occurred.

ungetc(fp, c)

File fp;
char c;

ungetc pushes one character back onto a file's buffer. Just one character can be pushed, and no error is reported if more than one is pushed--previously pushed characters are lost.

Returns **char:**

c.

ungetchar(c)

char c;

ungetchar is defined to be **ungetc(stdin, c).**

Returns **char:**

c.

unlink("name")
or
unlink(fname)

char *fname;

unlink deletes the file with the name **name** from the file system.

Returns pointer to **char:**

fname if ok;
NULL if access error or if file does not exist.

write(fn, buf, n)

int fn;
char *buf;
int n;

write writes **n** characters from **buf** onto the file with file number **fn** and is implemented as described in the reference text.

Returns **int:**

number of bytes written;
-1 if an error occurred.

Chapter 5

MISCELLANEOUS FUNCTIONS

This chapter lists functions which are available in the Cromemco C library to perform useful tasks. **move**, **strcpy**, **fill**, and **zap** have been written in assembly language for speed and space efficiency.

alloc(nbytes)

unsigned nbytes;

alloc obtains a block of memory **nbytes** long from the memory available for use by the program. See section 8.4, Memory Usage, for a discussion of the effects that a call to **alloc** has on the top-of-memory pointer, **_himem**, and how to restore memory to the state which existed before the first call to **alloc.**

alloc is declared in the **stdio.h** header file as:

char *alloc();

Returns pointer to **char**:

pointer to first byte of allocated block;
NULL if a block of the requested size was not available.

ccdos(z80regs)

union _regs z80regs;

ccdos executes the CDOS system call indicated by the value of the c register field in the union **z80regs. z80regs** is a union of register structures in which is stored the values to be loaded into the actual Z80 registers by **ccdos.** The header file **cdoscalls.h** contains definitions for all of the allowed CDOS system calls, and the header file **z80regs.h** contains the declaration of the union **_regs** which includes convenient field declarations for single, double, and double pair registers. Values returned in the registers by CDOS are loaded into the appropriate fields in **z80regs.**

See Chapter 7, Standard Header Files, for listings of the header files, Chapter 6, System Calls, for more information about system calls, and the CDOS instruction manual, Cromemco part number 023-0036, for descriptions of the system calls.

ccromix(SYSCALL, z80regs)

```
int SYSCALL;
union _regs z80regs;
```

ccromix executes the Cromix operating system call with the number **SYSCALL.** **z80regs** is a union of register structures in which is stored the values to be loaded into the actual Z80 registers by **ccromix.** The header files **jsysequ.h** and **modeequ.h** contain definitions for all of the allowed Cromix system calls and convenient definitions of mode numbers and masks, and the header file **z80regs.h** contains the declaration of the union **_regs** which includes convenient field declarations for single, double, and double pair registers. Values returned in the registers by the Cromix operating system are loaded into the appropriate fields in **z80regs.**

See Chapter 7, Standard Header Files, for listings of the header files, Chapter 6, System Calls, for more information about system calls, and the Cromix instruction manual, Cromemco part number 023-4022, for descriptions of the system calls.

Returns **int:**

0 if no error occurred;
non-zero if an error occurred;

exit(status)

```
int status;
```

exit aborts the program and returns to the operating system with the value in **status.** C will flush the buffers of all open files and close the files. **status** has a range of 0-255 when used in a program compiled for CDOS, and a range 0-65,535 when used in a program compiled to run on a Cromix system.

fill(dest, length, c)

```
char *dest;
unsigned length;
char c;
```

fill fills **length** bytes of memory with the character value **c** starting at the address **dest.**

free(pchar)

char *pchar;

free returns the block of memory pointed to by **pchar** to the free list. The block must have been one originally given to the program by **alloc** in order for C to consolidate the block with any unallocated adjacent blocks. See section 8.4, Memory Usage, for more details.

free is declared in the **stdio.h** header file as:

unsigned free;

Returns **unsigned:**

size of the consolidated block of memory;
0 if the block was not originally obtained by a call to **alloc.**

isalpha(c)

char c

isalpha has a value of 1 if **c** is an upper or lower case letter, otherwise it has a value of 0. This is a macro, defined in **stdio.h** and **cdstdio.h,** and can have a side effect when used with a pre- or post- incremented or decremented expression.

isdigit(c)

char c

isdigit has a value of 1 if **c** is a digit in the range 0-9, otherwise it has a value of 0. This is a macro, defined in **stdio.h** and **cdstdio.h,** and can have a side effect when used with a pre- or post- incremented or decremented expression.

islower(c)

char c

islower has a value of 1 if **c** is a lower case letter, otherwise it has a value of 0. This is a macro, defined in **stdio.h** and **cdstdio.h,** and can have a side effect when used with a pre- or post- incremented or decremented expression.

isspace(c)

char c

isspace has a value of 1 if **c** is a space, carriage return, line feed (newline), or tab character, otherwise it has a value of 0. This is a macro, defined in **stdio.h** and **cdstdio.h**, and can have a side effect when used with a pre- or post- incremented or decremented expression.

isupper(c)

char c

isupper has a value of 1 if **c** is an upper case letter, otherwise it has a value of 0. This is a macro, defined in **stdio.h** and **cdstdio.h**, and can have a side effect when used with pre- or post- incremented or decremented expressions.

max(a, b)

max evaluates to the maximum of **a** and **b.** This is a macro, defined in **stdio.h** and **cdstdio.h**, and can have a side effect when used with pre- or post- incremented or decremented expressions.

min(a, b)

min evaluates to the minimum of **a** and **b.** This is a macro, defined in **stdio.h** and **cdstdio.h**, and can have a side effect when used with a pre- or post- incremented or decremented expression.

move(to, from, length)

char *to;
char *from;
unsigned length;

move does what its name implies--it moves **length** characters from one character array to another.

rcdos(bc, de, hl)

```
unsigned bc, de, hl;
```

rcdos provides a somewhat more convenient way to execute CDOS system calls than does **ccdos.** The arguments **bc, de,** and **hl** contain the values to be inserted into the BC, DE, and HL registers prior to the execution of the CDOS call. The least significant byte of **bc** will be inserted into the C register, the most significant into the B register, and so on for **de** and **hl.** The value returned to **rcdos** in the A register by CDOS is returned as an integer from **rcdos.** This function has the disadvantage that it returns only one value from CDOS, unlike **ccdos,** which returns the values from all the registers.

Returns **int:**

value from the A register.

resetmem( )

resetmem restores memory between the program and the operating system to its initial configuration, unconditionally freeing all allocated blocks of memory. This function should not be used while any files are open for buffered I/O. See Section 8.4, Memory Usage, for a complete discussion of the way in which memory is allocated and freed.

Returns:

nothing

strcpy(to, from)

```
char *to;
char *from;
```

strcpy copies a string from one character array to another until it finds a null character. The null character is copied.

syserr("string")

syserr writes **string** to stderr, closes all files, and aborts the program with an exit code value of 255. C will flush the buffers of all open files and close the files.

system("string")

```
char *string;
```

system forks a shell process to execute the shell command string pointed to by **string** and waits for the forked shell to terminate. This function is available only to programs linked to execute under the Cromix operating system.

tolower(c)

```
char c
```

tolower converts **c** to lower case if it is an upper case letter, otherwise it does nothing to **c.** This is a macro, defined in **stdio.h** and **cdstdio.h,** and can have a side effect when used with a pre- or post- incremented or decremented expression.

toupper(c)

```
char c
```

toupper converts **c** to upper case if it is a lower case letter, otherwise it does nothing to **c.** This is a macro, defined in **stdio.h** and **cdstdio.h,** and can have a side effect when used with a pre- or post- incremented or decremented expression.

zap(dest, length)

```
char *dest;
unsigned length;
```

zap zero fills **length** bytes of memory starting at the address **dest.**

Chapter 6

SYSTEM CALLS

This chapter describes the formats of both CDOS and Cromix system calls. A complete discussion of each CDOS call can be found in the CDOS Manual, part number 023-0036; Cromix calls are discussed in the Cromix Operating System Manual, part number 023-4022.

6.1 CDOS Calls

A user program can execute a CDOS system call by using either the **rcdos** function or the **ccdos** function described in chapter 5 of this manual. A program which calls **ccdos** must first load the desired values of the Z80 registers into a union of the type _regs, which is declared for convenience in the **z80regs.h** header file. Chapter 7 of this manual contains a listing of this file.

Remember that **rcdos** returns as an integer value the value returned to it from CDOS in the A register. It does not load the values of the other registers into the register union. **ccdos** does load the values of the registers into the register union for the use of the calling program.

The program **cdcalls.c** on the distribution diskette illustrates the use of CDOS calls to open, read, and close a file.

6.2 Cromix Calls

There are two ways to execute Cromix system calls using Cromemco C.

First, a user program can use the **ccromix** function described in chapter 5 of this manual. A program which calls this function must first load the desired values of the Z80 registers into a union of the type _regs, which is declared for convenience in the **z80regs.h** header file. Chapter 7 of this manual contains a listing of this file.

Many Cromix calls require or return a channel number. This channel number is identical with the file number used with the I/O functions listed in Chapter 4, Input and Output.

The program **calls.c** on the distribution diskette illustrates the use of Cromix calls to open, read, and close a file.

Second, the program can call interface functions, provided in the Cromix C function library, **clib.rel**, which execute the system calls after loading the argument values into the appropriate registers. These functions have the same names, less the starting periods, as the system calls which they execute. For example, the function which calls the **.getdir** system call is named **getdir**.

Many of the system calls return values, and some load values into buffers provided to them by the calling program. The interface functions coordinate the return of values so that the C program can use those values just as it does those returned from C functions.

When a system call reports an error, the interface function returns **int** -1 to the calling C program after saving the values of the Cromix error code, the **de** register pair, and the **hl** register pair. These values are loaded into external variables named, respectively, **errno, errde,** and **errhl.** A C program which needs to access the error code or the register values must declare the variables as external:

```
extern int errno, errde, errhl;
```

Note that several Cromix calls return values whose type depends on what the programmer has requested. The C programmer must be certain to declare in his program that the function is returning a value of the desired type, otherwise the compiler will not generate correct code. For example, the **fstat** function can return a pointer to a character buffer, an **int,** or a **long,** depending on what the programmer has requested. A C function which uses **fstat** to obtain the size of a file must declare **fstat** to be **long:**

```
long fstat();
```

The following pages of this chapter list the available system call functions and the types of the required arguments. Cromix system calls which have the same name as functions or system calls defined in the reference text, such as **open,** are represented by functions which behave as described in the reference text, not by one of these system call functions.

caccess(channel, mask)

```
int channel
int mask
```

caccess tests channel access.

returns:

0 if the caller has the indicated access
or
-1 if not

cchstat(channel, statustype, statusvalue)

```
int channel
int statustype
int statusvalue
```

This form of **cchstat** is used to change the owner and group of a channel.

OR

cchstat(channel, statustype, statusvalue, statusmask)

```
int channel
int statustype
int statusvalue
int statusmask
```

This form of **cchstat** is used to change the access privileges of a channel.

OR

cchstat(channel, statustype, statustime)

```
int channel
int statustype
char statustime[6]
```

This form of **cchstat** is used to change one of the times associated with a channel.

returns:

int 0
or
-1 if error, and **int** error code in errno

chdup(channel)

int channel

chdup duplicates a channel.

returns:

int newchannel
or
-1 if error, and **int** error code in errno

chkdev(dtype, dnumber)

int dtype
int dnumber

chkdev checks for the presence of a device driver.

returns:

int 0
or
-1 if error, and **int** error code in errno

clink(channel, pathname)

int channel
char *pathname

clink establishes a link to an open file.

returns:

int 0
or
-1 if error, and **int** error code in errno

close(fn)

int fn;

close closes an open file. This **close** is identical to the **close** described in Chapter 4, Input and Output.

create(filename, accessmode, accessmask)

char *filename
int accessmode
int accessmask

create creates and opens a new file.

returns:

int channel
or
-1 if error, and **int** error code in errno

cstat(channel, statustype, buffer)

int channel
int statustype
char buffer[128]

cstat determines the status of a file which is open.

returns:

the full 128-byte inode, loaded into **buffer**;

or

one of:
int owner, group, owneraccess, groupaccess, otheraccess, nlinks, inodenumber, devicenumber;

or

int devicetypeandnumber;

or

long filesize;

or

the time created, time modified, time last accessed, or time last dumped, loaded into the first six bytes of **buffer**; the bytes contain binary numbers in the order year, month, day, hours, minutes, seconds;

or

-1 if error, and **int** error code in errno

cxexit(status)

int status

cxexit implements the **exit** Cromix system call; it exits from a program.

returns:

nothing

cxopen(filename, accessmode, accessmask)

char *filename
int accessmode
int accessmask

cxopen implements the **open** Cromix system call; it opens a file.

returns:

int channel
or
-1 if error, and **int** error code in errno

delete(filename)

char *filename

delete deletes a file from the file system.

returns:

int 0
or
-1 if error, and **int** error code in errno

error(channel)

int channel

error displays the error message defined by the Cromix system for the value of **errno**, which was loaded the last time a Cromix call returned an error.

returns:

int 0
or
-1 if error, and **int** error code in errno

exec(pathname, argv)

```
char *pathname
char *argv[]
```

exec executes a program.

returns:

nothing
or
-1 if error, and **int** error code in errno

exit - implemented as **cxexit**

faccess(pathname, mask)

```
char *pathname
int mask
```

faccess tests file access.

returns:

0 if the caller has the indicated access
or
-1 if not

fchstat(pathname, statustype, statusvalue)

```
char *pathname
int statustype
int statusvalue
```

This form of **fchstat** is used to change the owner and group of a file.

OR

fchstat(pathname, statustype, statusvalue, statusmask)

```
char *pathname
int statustype
int statusvalue
int statusmask
```

This form of **fchstat** is used to change the access privileges of a file.

OR

fchstat(pathname, statustype, statustime)

> **char** *pathname
> **int** statustype
> **char** statustime[6]

This form of **fchstat** is used to change one of the times associated with a file.

returns:

> **int** 0
> or
> -1 if error, and **int** error code in errno

fexec(pathname, chmask, argv)

> **char** *pathname
> **int** chmask
> **char** *argv[]

fexec forks and executes a program.

returns:

> **int** childprocessnumber
> or
> -1 if error, and **int** error code in errno

flink(pathname, newpathname)

> **char** *pathname
> **char** *newpathname

flink establishes a link to a file.

returns:

> **int** 0
> or
> -1 if error, and **int** error code in errno

fshell(argv)

char *argv[]

fshell forks a Shell process.

returns:

int processid
or
-1 if error, and **int** error code in errno

fstat(pathname, statustype, buffer)

char *pathname
int statustype
char buffer[128]

fstat determines the status of a file.

returns:

the full 128-byte inode, loaded into **buffer**;

or

one of:
int owner, group, owneraccess, groupaccess, otheraccess, nlinks, inodenumber, devicenumber;

or

int devicetypeandnumber; high byte = type, low byte = number;

or

long filesize;

or

the time created, time modified, time last accessed, or time last dumped, loaded into the first six bytes of **buffer**; the bytes contain binary numbers in the order year, month, day, hours, minutes, seconds;

or

-1 if error, and **int** error code in errno

getdate(date)

> **char** date[4]

getdate gets the date.

returns:

> the month, day, year, and day of week, loaded into **date** as binary numbers, in that order

getdir(buffer)

> **char** buffer[128]

getdir determines the pathname of the current directory.

returns:

> pathname of the current directory, loaded into **buffer**

getgroup(idtype)

> **int** idtype

getgroup gets the group id.

returns:

> **int** groupid

getmode(channel, modenumber)

> **int** channel
> **int** modenumber

getmode gets the characteristics of a console device.

returns:

> **int** mode
> or
> -1 if error, and **int** error code in errno

getpos(channel)

int channel

getpos gets the value of the file pointer.

returns:

long filepointer
or
-1 if error, and **int** error code in errno

getproc( )

getproc gets the process id.

returns:

int processid

gettime(time)

char time[3]

gettime gets the time.

returns:

the current hours, minutes, and seconds, loaded into **time** as binary numbers, in that order

getuser(idtype)

int idtype

getuser gets the user id.

returns:

int userid

indirect - identical to **ccromix,** described in Chapter 5.

makdev(pathname, dtype, dnumber)

> **char** *pathname
> **int** dtype
> **int** dnumber

makdev creates a new name for a device.

returns:

> nothing

makdir(pathname)

> **char** *pathname

makdir creates a new directory.

returns:

> nothing

mount(dummypathname, devpathname, access)

> **char** *dummypathname
> **cha**
> ar *devpathname
> **int** access

mount enables access to a file system.

returns:

> **int** 0
> or
> -1 if error, and **int** error code in errno

open - implemented as **cxopen**

rdbyte(channel)

int channel

rdbyte reads one byte.

returns:

int nextbyte
or
-1 if error, and **int** error code in errno

rdline(channel, buffer, maxbytes)

int channel
char buffer[]
int maxbytes

rdline reads one line. **buffer** must be large enough to hold the largest expected line.

returns:

int numberofbytesread
or
-1 if error, and **int** error code in errno

rdseq(channel, buffer, bytecount)

int channel
char buffer[]
int bytecount

rdseq reads bytes sequentially. **buffer** must be at least **bytecount** bytes in size.

returns:

int numberofbytesread
or
-1 if error, and **int** error code in errno

setdate(date)

```
char date[3]
```

setdate sets the date to the binary values of month, day, and year in **date.**

returns:

    nothing

setdir(pathname)

```
char *pathname
```

setdir changes the current directory.

returns:

    int 0
    or
    -1 if error, and int error code in errno

setgroup(idtype, idvalue, idnumber)

```
int idtype
int idvalue
int idnumber
```

setgroup sets the group id.

returns:

    nothing

setmode(channel, modenumber, modevalue, modemask)

```
int channel
int modenumber
int modevalue
int modemask
```

setmode sets the characteristics of a character device.

returns:

    int oldmode
    or
    -1 if error, and int error code in errno

setpos(channel, filepointer, mode)

> **int** channel
> **long** filepointer
> **int** mode

setpos sets the value of the file pointer. If **filepointer** is a constant, it must be explicitly **long**, e. g. 0L, 75L.

returns:

> **int** 0
> or
> -1 if error, and **int** error code in errno

settime(time)

> **char** time[3]

settime sets the time.

returns:

> nothing

setuser(idtype, idvalue, idnumber)

> **int** idtype
> **int** idvalue
> **int** idnumber

setuser changes the user id. The symbolic constant **id_hl**, defined in **jsysequ.h**, indicates that the new id is found in **idnumber**.

returns:

> nothing

shell(argv)

char *argv[]

shell transfers control to a Shell process.

returns:

nothing

trunc(channel)

int channel

trunc truncates an open file.

returns:

int 0
or
-1 if error, and **int** error code in errno

unmount(devpathname, ejectflag)

char *devpathname
int ejectflag

unmount disables access to a file system.

returns:

int 0
or
-1 if error, and **int** error code in errno

update( )

update updates the disk buffers.

returns:

int 0
or
-1 if error, and **int** error code in errno

version( )

version gets the system version number as a binary coded decimal value.

returns:

int version

wait(flag, childid, statuses)

int flag
int childid
int statuses[2]

wait waits for the termination of a child process.

returns:

int childid, and loads the process and system termination statuses into **statuses**
or
-1 if error, and **int** error code in errno

wrbyte(channel, byte)

int channel
char byte

wrbyte writes one byte.

returns:

int 0
or
-1 if error, and **int** error code in errno

wrline(channel, buffer)

int channel
char buffer[]

wrline writes one line.

returns:

int numberofbyteswritten
or
-1 if error, and **int** error code in errno

wrseq(channel, buffer, bytecount)

```
int channel
char buffer[]
int bytecount
```

wrseq writes sequentially.

returns:

int numberofbyteswritten
or
-1 if error, and **int** error code in errno

Chapter 7

STANDARD HEADER FILES

This chapter contains listings of the standard header files supplied with Cromemco C.

The file **stdio.h** must be included in all programs which use functions from the Cromix C function library, and the file **cdstdio.h** must be included in programs which use functions from the CDOS C function library. The file **z80regs.h** contains a register union declaration and definitions for single and double registers and double register pairs which are used to pass register values to the **ccromix, ccdos,** and **rcdos** functions.

The files **jsysequ.h, modeequ.h,** and **cdoscalls.h** contain convenient definitions for Cromix system call numbers and CDOS system call numbers.

7.1 cdoscalls.h

```
/* cdoscalls.h:  Cromemco C I/O header file

   Copyright (c) 1980 by Cromemco, Inc., All Rights Reserved

   This file contains definitions for all CDOS system calls
   which can be made using the functions rcdos and _ccdos */


#define CDOSABORT       0
#define CDOSRDCONS      1
#define CDOSWRCONS      2
#define CDOSRDRDR       3
#define CDOSWRPUN       4
#define CDOSWRLPT       5
#define CDOSGETIOB      7
#define CDOSSETIOB      8
#define CDOSPUTL        9
#define CDOSGETL        10
#define CDOSTESTC       11
#define CDOSRELEASE     12
#define CDOSBOOT        13
#define CDOSSELECT      14
#define CDOSOPEN        15
#define CDOSCLOSE       16
#define CDOSERASE       19
#define CDOSREAD        20
#define CDOSWRITE       21
#define CDOSCREATE      22
#define CDOSRENAME      23
#define CDOSLOG         24
#define CDOSQDSK        25
#define CDOSDMA         26
#define CDOSRDNOECHO    128
#define CDOSSETCC       130
#define CDOSRDBLK       131
#define CDOSWRBLK       132
#define CDOSBFN         134       /* note:  address of terminator
                                            not returned */
#define CDOSUPDATE      135
#define CDOSLINK        136
#define CDOSMUL         137
#define CDOSDIV         138
#define CDOSHOME        139
#define CDOSEJECT       140
#define CDOSCRT         142
#define CDOSSETDATE     143
#define CDOSSETTIME     145
#define CDOSRCODE       147
#define CDOSATTR        148
#define CDOSBOTTOM      151
```

7.2 cdstdio.h

```
/* cdstdio.h:  Cromemco C I/O header file for use with CDOS C library

   Copyright (c) 1980, 1981 by Cromemco, Inc., All Rights Reserved

   This file contains declarations of values used in conjunction
   with functions from the Cromemco CDOS C library.
*/

/***** I/O related definitions *****/

typedef int     FILE;
typedef int     File;

#define EOF     (-1)
#define ERR     (-1)
#define CDOSEOF 26              /* EOF on CDOS ascii files is CNTRL-Z */

#define NULL    0
#define READ    0
#define WRITE   1
#define UPDATE  2

#define STDIN   0
#define STDOUT  1
#define STDERR  2
#define stdin   0
#define stdout  1
#define stderr  2

#define backc(fn,x) ungetc(fn,x)
#define close(x) _cclose(x)
#define fseek lseek                     /* fseek same as lseek under CDOS */
#define getchar(x) getc(stdin)
#define putchar(x) putc(x,stdout)
#define ungetchar(x) ungetc(stdin,x)
#define unlink(x) _cerase(x)


/***** non-integer function declarations *****/

char    *alloc();
char    *fgets();
char    *fputs();
File    fopen();
File    fclose();
unsigned free();
long    lseek();


/* commonly used macros;  caution:  these macros can
   cause side effects when passed a pre- or
   post-decremented character expression */

#define isdigit(c)      ( '0'<=(c) && (c)<='9' )
#define isupper(c)      ( 'A'<=(c) && (c)<='Z' )
#define islower(c)      ( 'a'<=(c) && (c)<='z' )
#define isalpha(c)      ( isupper(c) || islower(c) )
#define isspace(c)      ( (c) == ' '  || (c) == '\n' || \
                          (c) == '\r' || (c) == '\t' )
#define toupper(c)      ( islower(c) ? (c) & 0x5f : (c) )
#define tolower(c)      ( isupper(c) ? (c) | 0x20 : (c) )
#define min(a,b)        ( (a) < (b) ? (a) : (b) )
#define max(a,b)        ( (a) > (b) ? (a) : (b) )
```

7.3 jsysequ.h

```
/* jsysequ.h:  Cromemco C I/O header file

   Copyright (c) 1980 by Cromemco, Inc., All Rights Reserved

   This file contains declarations of all values which are
   used during calls to the Cromix operating system.
*/

/*
   access modes for create
*/

#define op_read          0       /* read only */
#define op_write         1       /* write only */
#define op_rdwr          2       /* read and write */
#define op_append        3       /* append only */
#define op_xread         4       /* exclusive read only */
#define op_xwrite        5       /* exclusive write only */
#define op_xrdwr         6       /* exclusive read and write */
#define op_xappend       7       /* exclusive append only */

#define op_truncf        0x80    /* truncate on create flag */
#define op_condf         0x40    /* conditional create flag */

/*
   status types for _fstat, _cstat, _fchstat, _cchstat
*/

#define st_all           0       /* all of inode (128 bytes) */
#define st_owner         1       /* owner */
#define st_group         2       /* group */
#define st_aowner        3       /* owner access, mask */
#define st_agroup        4       /* group access, mask */
#define st_aother        5       /* other access, mask */
#define st_ftype         6       /* file type, special device # */
#define st_size          7       /* file size */
#define st_nlinks        8       /* number of links */
#define st_inum          9       /* inode number */
#define st_device        10      /* device containing inode */
#define st_tcreate       11      /* time created */
#define st_tmodify       12      /* time last modified */
#define st_taccess       13      /* time last accessed */
#define st_tdumped       14      /* time last dumped */

/*
    file types for st_ftype
*/

#define is_ordin         0       /* ordinary file */
#define is_direct        1       /* directory file */
#define is_char          2       /* character device */
#define is_block         3       /* block device */

/*
    mask values for file access flags
*/

#define ac_read          0x01    /* read access bit */
#define ac_exec          0x02    /* execute access bit */
#define ac_writ          0x04    /* write access bit */
#define ac_apnd          0x08    /* append access bit */

/*
   id types and values for _setuser, _getuser, _setgroup, _getgroup
*/

#define id_effective     0       /* effective id */
#define id_login         1       /* login id */
#define id_program       2       /* program id */
#define id_hl            3       /* id contained in idnumber */

/*
    set position modes
```

```
*/

#define pos_begin       0          /* beginning of file */
#define pos_current     1          /* current position of file */
#define pos_end         2          /* end of file */

/*
    Cromix System Call Numbers
*/

#define _makdev         0x00       /* make device entry */
#define _makdir         0x01       /* make a directory */
#define _getdir         0x02       /* get current directory name */
#define _setdir         0x03       /* change current directory */

#define _mount          0x04       /* mount file system */
#define _unmount        0x05       /* unmount file system */
#define _delete         0x06       /* delete file */
#define _chkdev         0x07       /* check for device driver */

#define _create         0x08       /* create & open file */
#define _open           0x09       /* open file */
#define _chdup          0x0A       /* duplicate channel */
#define _close          0x0B       /* close file */

#define _trunc          0x0D       /* truncate open file */

#define _getpos         0x10       /* get file position */
#define _setpos         0x11       /* set file position */
#define _getmode        0x12       /* get device characteristics */
#define _setmode        0x13       /* set device characteristics */

#define _rdseq          0x14       /* read n bytes */
#define _wrseq          0x15       /* write n bytes */
#define _rdbyte         0x16       /* read 1 byte */
#define _wrbyte         0x17       /* write 1 byte */
#define _rdline         0x18       /* read a line */
#define _wrline         0x19       /* write a line */
#define _printf         0x1B       /* print formatted string */
#define _error          0x1C       /* print error message */

#define _fstat          0x20       /* get file status (inode) */
#define _cstat          0x21       /* get channel status (inode) */
#define _fchstat        0x22       /* change file status */
#define _cchstat        0x23       /* change channel status */
#define _flink          0x24       /* link to file */
#define _clink          0x25       /* link to open channel */
#define _faccess        0x26       /* test file access */
#define _caccess        0x27       /* test channel access */

#define _getdate        0x30       /* get date */
#define _setdate        0x31       /* set date */
#define _gettime        0x32       /* get time */
#define _settime        0x33       /* set time */

#define _getuser        0x34       /* get user id */
#define _setuser        0x35       /* set user id */
#define _getgroup       0x36       /* get group id */
#define _setgroup       0x37       /* set group id */

#define _getproc        0x3A       /* get process id */
#define _wait           0x45       /* wait for child process */
#define _exit           0x46       /* exit process (close files) */

#define _fshell         0x48       /* fork a shell process */
#define _shell          0x49       /* transfer to shell process */
#define _fexec          0x4B       /* fork and execute program */
#define _exec           0x4C       /* execute program */

#define _indirect       0x51       /* system call in A-register */

#define _update         0x52       /* update disk I/O buffers */
#define _mult           0x53       /* multiply */
#define _divd           0x54       /* divide */
#define _version        0x55       /* get system version # */
#define _boot           0x56       /* boot new operating system */

/*
        Cromix error numbers returned in extern int errno
```

```
*/

#define _badchan        1       /* bad channel # */
#define _toomany        2       /* channel already open */
#define _notopen        3       /* channel not open */
#define _endfile        4       /* end-of-file */
#define _ioerror        5       /* I/O error */
#define _filtable       6       /* file table exhausted */
#define _notexist       7       /* file does not exist */
#define _badname        8       /* bad file name */
#define _diraccess      9       /* directory access */
#define _filaccess      10      /* file access */
#define _exists         11      /* file already exists */
#define _nospace        12      /* no disk space left */
#define _noinode        13      /* no inodes left */
#define _inotable       14      /* inode table exhausted */
#define _badcall        15      /* illegal system call */
#define _filsize        16      /* file size too big */
#define _mnttable       17      /* mount table exhausted */
#define _notdir         18      /* not a directory */
#define _isdir          19      /* is a directory */
#define _priv           20      /* privileged system call */
#define _notblk         21      /* not a block special device */
#define _fsbusy         22      /* file system busy */
#define _notordin       23      /* not an ordinary file */
#define _notmount       24      /* device not mounted */
#define _nochild        25      /* no child processes */
#define _nomemory       26      /* not enough memory */
#define _ovflo          27      /* divide overflow */
#define _argtable       28      /* argument table exhausted */
#define _arglist        29      /* bad argument list */
#define _numlinks       30      /* too many number of links */
#define _difdev         31      /* cross-device link */
#define _nodevice       32      /* no special device */
#define _usrtable       33      /* user process table exhausted */
#define _badvalue       34      /* value out of range */
#define _notconn        35      /* I/O device not connected */
#define _keybaud        36      /* set baud rate from the keyboard */
#define _diruse         37      /* directory in use (delete) */
#define _filuse         38      /* file in use (exclusive access) */
#define _nomatch        39      /* no match on ambiguous name */
#define _chnaccess      40      /* channel access */
#define _notcromix      41      /* not a cromix disk */

#control list
```

7.4 modeequ.h

```
/* modeequ.h:  Cromemco C I/O header file

   Copyright (c) 1980 by Cromemco, Inc., All Rights Reserved

   This file contains declarations of all values which are
   used in the getmode and setmode Cromix system calls.
*/

/*
    _SETMODE & _GETMODE mode numbers
*/

#define md_iwake        0        /* input wakeup */
#define md_ibaud        0        /* input baudrate (see below) */
#define md_owake        1        /* output wakeup */
#define md_obaud        1        /* output baudrate (see below) */
#define md_model        2        /* model (see below) */
#define md_mode2        4        /* mode2 (see below) */
#define md_mode3        5        /* mode3 */
#define md_erase        6        /* newvalue = auxilliary erase character */
#define md_dlecho       7        /* newvalue = char echoed when the del, erase,
                                    or ^H key is pushed.  (if this character is
                                    'R' or 'r', backspace, space, & backspace
                                    are echoed.) */
#define md_kill         8        /* newvalue = linekill character */
#define md_signal       9        /* newvalue = user signal character (when the
                                    md2_sgenable bit of mode2 is set) */
#define md_width        11       /* newvalue = page width (when md2_wrapenable
                                    of mode2 is set) */
#define md_length       10       /* newvalue = page length (when md2_pgenable
                                    of mode2 is set) */
#define md_bmargin      12       /* newvalue = page bottom margin ( when
                                    md2_pgenable of mode2 is set) */
#define md_nlnulls      15       /* newvalue = number of nulls sent after a nl
                                    */
#define md_tabnulls     16       /* newvalue = number of nulls sent after a tab
                                    */
#define md_ffnulls      17       /* newvalue = number of nulls sent after a ff
                                    */
#define md_crnulls      18       /* newvalue = number of nulls sent after a cr
                                    */
#define md_status       13       /* channel status (_GETMODE only) (see below)
                                    */
#define md_discard      13       /* discard input (_SETMODE only) (see below) */
#define md_ident        14       /* channel identification (_GETMODE only)
                                    (below) */
#define md_noway        (-1)     /* illegal call number

/*
   byte contents of newvalue for md_ibaud & md_obaud calls:
*/

#define b_hangup        0        /* hang up dataphone */
#define b_50            1        /* 50 baud */
#define b_75            2        /* 75 baud */
#define b_110           3        /* 110 baud */
#define b_134           4        /* 134.5 baud */
#define b_150           5        /* 150 baud */
#define b_200           6        /* 200 baud */
#define b_300           7        /* 300 baud */
#define b_600           8        /* 600 baud */
#define b_1200          9        /* 1200 baud */
#define b_1800          10       /* 1800 baud */
#define b_2400          11       /* 2400 baud */
#define b_4800          12       /* 4800 baud */
#define b_9600          13       /* 9600 baud */
#define b_exta          14       /* External A */
#define b_extb          15       /* External B */
#define b_19200         16       /* 19200 baud */

/*
   The following code signifies that CRs are to be input from keyboard
   to set the baudrate.
*/

#define b_auto          17
#define maxserspeed     17
```

```
/*
    The following code is used to merely wakeup interrupts.
    Neither the baudrate nor the baudrate code stored in the
    channel will be changed.
*/

#define b_wakeup        253

/*
    The following code  is used to wakeup interrupts_  The baudrate will
    not be changed, but the code will be stored in the channel.
*/

#define b_nochg         255
#define b_noway         254      /* illegal baud code (used to mark table
                                    start) */

/*
    mask values for modevalue for md_model mode:
    (the values set in modemask determine which values are used from
    modevalue.)
*/

#define mdl_hangup      0x01     /* hangup dataset after last close */
#define mdl_tab         0x02     /* software tabs [expand as spaces] */
#define mdl_lcase       0x04     /* map upper to lower case on input */
#define mdl_echo        0x08     /* echo (full duplex) */
#define mdl_cr_nl       0x10     /* on input, map cr into nl (lf), & echo lf or
                                    cr as crlf */
#define mdl_raw         0x20     /* raw mode: wake up on all characters */
#define mdl_odd         0x40     /* odd parity allowed on input */
#define mdl_even        0x80     /* even  "          "     "     " */

/*
    masks for modevalue and modemask for md_mode2 calls:
    (the values set in modemask determine which values are used from
    modevalue.)
*/

#define md2_pause       0x01     /* after 24 lines output, wait for cntrl-Q */
#define md2_later       0x02     /* wait until character is used before echoing
                                    it */
#define md2_noecnl      0x04     /* no echoing of line terminators */
#define md2_sgenable    0x08     /* user signal (md_sigchar) enable */
#define md2_abenable    0x10     /* cntrl-C abort enable */
#define md2_ff          0x20     /* software formfeeds [expand as newlines] */
#define md2_wrap        0x40     /* software wrap-around [insert newline when
                                    page width (md_width) is exceeded] */
#define md2_outraw      0x80     /* raw output:  don't process cr's, tab's,
                                    etc. */

/*
    masks for modevalue and modemask for md_mode3 calls:
    (the values set in modemask determine which values are used from
    modevalue.)
*/

#define md3_esccr       0x01     /* make ESC return lines */

/*
    masks of legal model & mode2 values for different types of channels:
*/

#define mdlv_tty        0xff             /* all attributes legal */
#define md2v_tty        0xff             /* all attributes legal */
#define mdlv_outp       mdl_tab
#define md2v_outp       (md2_ff | md2_wrap | md2_outraw)
#define mdlv_inp        (mdl_lcase | mdl_echo | mdl_cr_nl | \
                         mdl_raw | mdl_odd | mdl_even)
#define md2v_inp        (md2_later | md2_noecnl | md2_sgenable | md2_abenable)

/*
    masks for modevalue for md_status & md_discard calls:
*/

#define st_charrdy      0x01     /* at least one input character is ready */
#define st_linerdy      0x80     /* at least one input line is ready */
```

```
/*
   the following 3 flags are reset every time the status is checked:
*/

#define st_keybd         0x04     /* keyboard touched since status last checked
                                     */
#define st_signal        0x20     /* input signal character received */
#define st_abort         0x40     /* input cntrl-C received */

/*
   masks for modevalue for md_ident calls:
*/

#define id_tty           0x01     /* tty channel (with its mate, makes a full
                                     i/o channel) */
#define id_output        0x02     /* output channel (or output half of a tty
                                     channel) */
#define id_serial        0x04     /* serial channel */
#define id_nochg         0x08     /* no characteristics (except baudrate) of the
                                     input channel or the output channel in
                                     which this bit is set can be changed. */

#control list
```

7.5 stdio.h

```
/* stdio.h:  Cromemco C I/O header file

   Copyright (c) 1981 by Cromemco, Inc., All Rights Reserved

   This version of stdio.h is for inclusion in programs
   to be run under the Cromix operating system */


/***** File I/O control block definition *****/

struct _iocb
        {                        /* definition of the structure tag '_iocb' */
        unsigned _flags ;        /* see individual bit definitions, below   */
        unsigned _fnum ;         /* file number, as returned from cromix    */
        int      _bufsz ;        /* size of buffer assigned to this file    */
        char   * _first ;        /* address of first byte of buffer         */
        char   * _next ;         /* address of next byte in buffer to access*/
        int      _count ;        /* count of characters remaining in buffer */
        unsigned _backup ;       /* used to hold a single 'ungetc' char     */
        unsigned _misc ;         /* reserved (makes _iocb 16 bytes long)    */
        } ;

typedef struct _iocb FILE ;      /* FILE is thus a structure of above form  */
typedef struct _iocb *File ;     /* File is a pointer to such a structure   */

/***** Definitions of individual bits of the '_flags' part of _iocb *****/

#define FLAGREAD        0x01     /* readable file                         */
#define FLAGWMODE       0x02     /* file opened in write mode             */
#define FLAGAPPEND      0x04     /* file opened in append mode            */
#define FLAGWRITE       0x06     /* writeable file, write or append mode */
#define FLAGABORT       0x80     /* abort on getc/putc error if set       */
#define FLAGBACK        0x40     /* '_backup' contains a char if set      */
#define FLAGINUSE     0xFFFF     /* i.e., if _flag != 0, iocb is in use   */

/***** I/O related definitions *****/

#define BLOCKSIZE 512            /* default buffer size for buffered I/O */
#define MAXFILES  20             /* maximum # of open buffered files     */

#define EOF      (-1)
#define ERR      (-1)

#define NULL     0

#define READ     0
#define WRITE    1
#define UPDATE   2

#define STDIN    0
#define STDOUT   1
#define STDERR   2

extern  File     stdin, stdout, stderr ;

#define backc(fp,x) ungetc(fp,x)
#define getchar(x) getc(stdin)
#define getline(buf,max) getl(stdin,buf,max)
#define putchar(x) putc(x,stdout)
#define ungetchar(x) ungetc(stdin,x)

/*****************************************************************************/
```

```
/* commonly used macros;  caution:  these macros can
   cause side effects when passed a pre- or
   post-decremented character expression */

#define isdigit(c)        ( '0'<=(c) && (c)<='9' )
#define isupper(c)        ( 'A'<=(c) && (c)<='Z' )
#define islower(c)        ( 'a'<=(c) && (c)<='z' )
#define isalpha(c)        ( isupper(c) || islower(c) )
#define isspace(c)        ( (c) == ' '  || (c) == '\n' || \
                            (c) == '\r' || (c) == '\t' )
#define toupper(c)        ( islower(c) ? (c) & 0x5f : (c) )
#define tolower(c)        ( isupper(c) ? (c) | 0x20 : (c) )
#define min(a,b)          ( (a) < (b) ? (a) : (b) )
#define max(a,b)          ( (a) > (b) ? (a) : (b) )

/***** declarations of library functions that do not return int *****/

extern  File     fopen() ;
extern  File     fclose() ;

extern  char     *fgets(), *fputs() ;

extern  long     lseek(), fseek() ;

extern  char     *alloc() ;
extern  unsigned free()  ;
```

7.6 z80regs.h

```
/* z80regs.h:  Cromemco C I/O header file

   Copyright (c) 1980 by Cromemco, Inc., All Rights Reserved

   This header file contains the structure and structure
   field definitions which permit convenient data transfer
   between user programs and the function _ccromix */


/***** declaration of the register structures *****/

struct  _rb     { char _rf, _ra, _rc, _rb, _rl, _rh, _re, _rd; } ;
struct  _rw     { unsigned _raf, _rbc, _rhl, _rde; } ;
struct  _rl     { long _rbcaf, _rdehl; } ;


/***** now the union declaration and data definition *****/

union _regs {
        struct _rb _rbytes ;
        struct _rw _rwords ;
        struct _rl _rlongs ;
        }  ;

static union _regs _r ;

#define rf      _r._rbytes._rf
#define ra      _r._rbytes._ra
#define rc      _r._rbytes._rc
#define rb      _r._rbytes._rb
#define rl      _r._rbytes._rl
#define rh      _r._rbytes._rh
#define re      _r._rbytes._re
#define rd      _r._rbytes._rd

#define raf     _r._rwords._raf
#define rbc     _r._rwords._rbc
#define rhl     _r._rwords._rhl
#define rde     _r._rwords._rde

#define rbcaf   _r._rlongs._rbcaf
#define rdehl   _r._rlongs._rdehl
```

Chapter 8

IMPLEMENTATION DETAILS

This chapter describes some of the details of the implementation of Cromemco C that might be useful to the user of C.

8.1 Data Types

The data types in Cromemco C have the following forms when stored in memory.

char one byte (8 bits) long. The most significant bit is not propagated when a **char** is converted to an **int.** The range of values of a **char** is 0 to 255.

short the same as **int.**

int two bytes (16 bits) long, stored LSB, MSB in the Z80 convention. The range of values of an **int** is -32768 to 32767.

unsigned the same as **int** except that the range of values is 0 to 65535.

long four bytes (32 bits) long, stored LSB,...,MSB. All **longs** are assumed to be signed. The range of values is approximately -2E9 to 2E9.

float four bytes (32 bits) long, with 6 digits of accuracy and values in the range +/- 9.99E-65 to +/- 9.99E+62. The four bytes are stored in the order:

> sign and base-10 exponent,
> most significant byte of mantissa,
> second byte of mantissa,
> least significant byte of mantissa.

The most significant bit in the first byte stores the sign; 0 indicates a positive number, 1 indicates a negative number. The exponent is stored in excess-64 (excess-40H) format (64 is added to the real exponent to form the stored exponent). The next 3 bytes contain the BCD (Binary Coded Decimal) mantissa, 2 digits to the byte, normalized to a value between .1 and 1. The implicit decimal point is located before the first digit of the mantissa.

double	eight bytes (64 bits) long with 14 digits of accuracy and values in the range +/- 9.99E-65 to +/- 9.99E+62. The first byte contains the sign and exponent as with **float**, and the next 7 bytes contain the 14 digit BCD mantissa, most significant byte first.

8.2 Argument Passing

When Cromemco C processes a function call of the form

```
function( arg1, arg2, ..., argN )
```

it generates code that will push the values of the actual arguments onto the machine stack in the order encountered and call the function using the assembler CALL instruction. The total number of bytes passed is stored as a one byte number in the next byte following the CALL instruction. Note that this method results in call by value: the item passed is the value of the argument, not the address of the argument.

The called function is expected to extract from the stack first the return address and then the arguments. The return address must be preserved and incremented by 1 before it is placed back onto the stack just prior to the return from the function. The stack organization is LIFO (last in first out), so the function extracts the value of the last argument in the list first, then the next-to-last value, and so on.

A method to use when interfacing an assembly language function with Cromemco C is described below.

The assembly language function should:

1. pop the return address to a 16-bit register pair;
2. obtain the argument byte count from the return address;
3. increment the return address by 1 to move it past the byte count;
4. pop all arguments as required;
5. push the return address (now correct) onto the stack;
6. execute the body of the function;
7. load the function return value into the appropriate register(s);
8. return.

The following two rules must be observed in order that the program execute correctly:

1. The assembly language routine must restore the IX register, used by C to access **auto** variables, to its entry value before returning.

2. The assembly language routine must pop all bytes of all arguments from the stack; if this is not done, the function which called the assembly language function will not return to the function which called it.

example:

An assembly language routine is called from C to output a byte to a port.

In the C program:

```
     .
     .
     .
     extern int out( int portnumber; int data; );

     portnumber = 0;
     data = 3f;
     out( portnumber, data );
```

In the assembly language routine:

```
        rel
        global  out
out:
        pop     hl      ; the return address
        ld      a,(hl)  ; argument byte count
        inc     hl      ; bump past byte count
        cmp     4       ; expecting 2 ints = 4 bytes
        jp      nz,???  ; not 4 bytes, jump to error handler
        pop     de      ; the data
        pop     bc      ; the port number is in c
        out     (c),e   ; output the byte
        push    hl      ; put return address back on stack
        ex      de,hl   ; hl = the data byte passed into this routi
        ret
```

example:

Arguments may also be passed to assembly language routines via global variables.

In the C program:

```
        extern value, a;
        value += 2;
        function();
        printf("%d", a);
        .
        .
        .
```

In the assembler routine:

```
        global  function, $$$a, value

$$$a:   dw      0
value:  dw      0

function:
        pop     hl        ; see section on function arguments
        ld      a,(hl)    ; number of argument bytes on stack
        or      a         ; is it 0? (we want no arguments)
        jp      nz,???    ; nz -> no, so jump to error routine
        inc     hl        ; set up real return address
        push    hl        ; on the stack
        ld      hl,(value)  ; 2 data bytes
        .
        .                 ; the body of the routine
        .
        ld      ($$$a),hl   ; load new value into the external
                            ; variable for use in the C program
        ret               ; back to the C program
```

8.3 Function Values

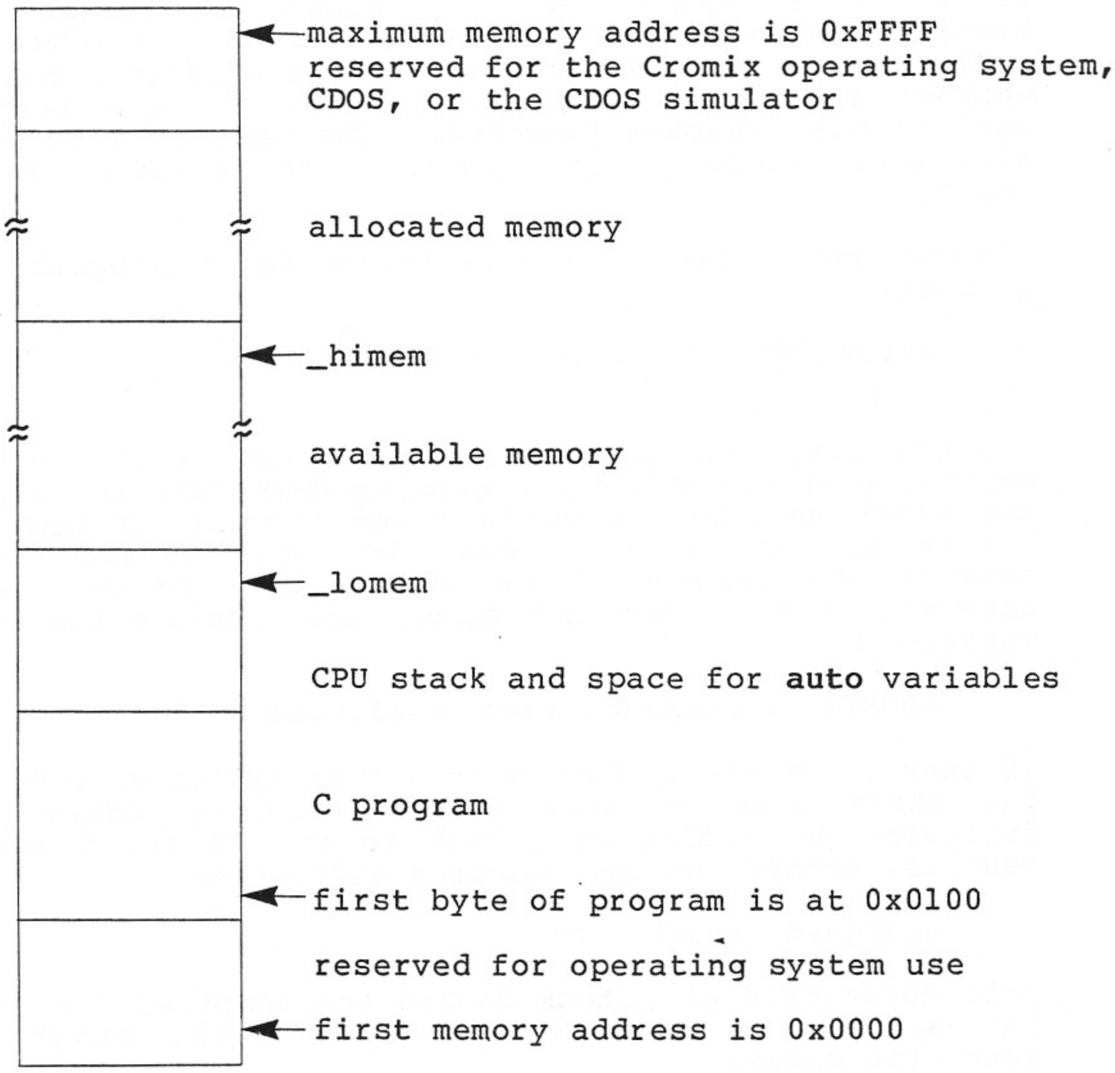
Cromemco C functions return their values in the Z80 registers:

> **char** functions return their values in the L register; the value of the H register is undefined;
>
> **int** and **unsigned** functions and functions which return pointers all return a 16 bit value in HL, most significant byte in H, least significant in L;
>
> **long** functions return 32 bits in the DEHL registers, most significant byte in D, byte 2 in E, byte 1 in H, and the least significant byte in L;
>
> **float** and **double** functions return their values in

the external variable **$FRO.**

8.4 Memory Usage

The following map illustrates how memory is used by a compiled C program and the operating system.


maximum memory address is 0xFFFF
reserved for the Cromix operating system, CDOS, or the CDOS simulator
allocated memory
_himem
available memory
_lomem
CPU stack and space for **auto** variables
C program
first byte of program is at 0x0100
reserved for operating system use
first memory address is 0x0000

At the start of the program **_lomem** is initialized to point to the first byte available above the CPU stack and **_himem** is initialized to point to the last byte available before the bottom of the operating system. When **alloc** is first called it saves the values of **_lomem** and **_himem** so that memory can at any time be restored to its initial state. If **alloc** cannot find a large enough unused block in allocated memory (memory between **_himem** and the bottom of the operating system) it will then look to available memory. If there's enough available memory to satisfy the request **alloc** will allocate the block to the C program and move **_himem** down by the size of the allocated block plus block header (3 bytes). The **free** function never moves **_himem** back up.

Memory can become fragmented, or checkerboarded, into unusably small blocks after repeated calls to **alloc** and **free**, and it might become necessary to restore the memory between **_lomem** and the bottom of the operating system to its original state. All of allocated memory, whether in use or not, can be freed in one step by calling the **resetmem** function. The next call to **alloc** will then return a block adjacent to operating system memory.

_lomem and **_himem** are available to C programs as globals:

```
     extern char *_lomem, *_himem;
```

The CPU stack and automatic space occupy a 2K block of memory, with the CPU stack growing down from the top of the block and the automatic space growing up from the bottom of the block. Each time one function calls another, the compiled C code checks that the two stacks have not crashed into each other, and displays the error message

```
     ABORT:  stack/auto space collision
```

if they do overlap. Should this overlap occur, redefine the stack size to some larger value by adding the following definition of **_stack** to one of the C source modules, outside of any function definition:

```
     unsigned _stack = n;
```

This definition of **_stack** causes the compiled C code to use an **n** byte block of memory for the stack and automatic space.

_stack is also defined in the C library, which is important to know when redefining the stack size: the Link program uses the first definition of an external name and ignores subsequent definitions. For this reason, one must instruct Link to load the C module containing the redefinition of **_stack** before it searches the C function library. See Section 9.5, Link, for more information about the linking process.

If the compiled and linked program with all of its functions is very large, the top of the stack can point into system memory. A C program will detect this before starting execution of any user code and will abort with the message:

```
ABORT:  program/stack too big
```

Should this happen, redefine **_stack** with a smaller size.

8.5 Register Usage

Cromemco C uses the L, HL, and DEHL registers to store the value of the last C expression:

a **char** value is left in L, H is 0;

an **int** value is left in HL;

an **unsigned** value is left in HL;

a pointer value is left in HL;

a **long** value is left in DEHL (D is MSB, L is LSB);

Expressions involving longs, floats, and function references use the prime registers.

Each C function uses the IX register to access its **auto** variables.

8.6 Programming Hints

Some constructs in Cromemco C generate more efficient code than others. This section lists those constructs which generate faster and smaller code than alternate constructs which do the same job.

1. references to **static** and **extern** variables are faster and take less code than references to **auto** variables.

2. **do...while()** generates faster and smaller code than any other looping construct.

3. Preincrement and predecrement are faster than postincrement and postdecrement.

4. **unsigned** expressions are evaluated faster than **int** expressions.

5. An expression which takes advantage of the fact that an assignment expression itself has a value will generate faster and more compact code that the equivalent in-line statements.

example:

```
     if ((c = getc(fp)) == (d = f())
```

generates more compact code than the equivalent in-line statements:

```
     c = getc(fp);
     d = f();
     if (c == d);
```

Chapter 9

USER'S GUIDE

The Cromemco C compiler consists of three separate passes which scan a C input program and generate assembly language output. The assembly language output is then assembled using the Cromemco Macro Assembler, which produces a relocatable object file. This relocatable file is not executable, and must be linked into executable form using the Cromemco Link program. The Link program produces an executable object file from one or more relocatable input files.

9.1 C Pass 0

Pass 0, called cp0, is the first pass to be executed when compiling a C program. cp0 is the preprocessor, which means that it processes all macro definitions, macro calls, and **#defines**, and produces an intermediate file for input into pass 1. cp0 is executed using a command line of the form

```
cp0 source intermediate1 options
```

where **source** names the C source file, **intermediate1** names the intermediate file, and **options** is an optional string of compiler options. The list of options follows.

<u>option</u>	<u>action</u>

-i	specifies that the next command line argument is the name of a directory to be searched when processing **#include "file"**. cp0 adds the argument to the front of **file** and then attempts to read the file with the new name. For example,
	
	`-i /usr/`
	
	and
	
	`#include "driver"`
	
	would cause cp0 to insert the contents of **/usr/driver** in place of the #include line.
	
	The default is the current directory.
	
	The argument may also be a drive identifier in the CDOS convention (e.g. A:) and the

compiler will add the drive identifier to the front of **file.** The CDOS simulator will convert the drive identifier to a directory name according to the translation conventions listed in section 4.4, Filenames.

-l same as -i except that it specifies the directory to be searched when processing **#include <file>.** The default is /usr/include/.

-p causes cp0 to display some pass 0 statistics when it finishes.

-s causes cp0 to include C source in the intermediate file. The source will not be inserted into the Z80 file output from cp2 unless -s is specified in the cp2 command line also. The -s option is equivalent to inserting **#control source** as the first line in the C program. This option is overridden by **#control nsource.**

9.2 C Pass 1

Pass 1, called cp1, is the second pass of the C compiler. cp1 contains the syntax checking and logical code generation portions of the compiler. cp1 is executed using a command line of the form

```
cp1 intermediate1 intermediate2
```

where **intermediate1** is the name of the intermediate file from cp0 and **intermediate2** is the name of the next intermediate file, used as input to cp2.

There are no compiler options for cp1.

9.3 C Pass 2

Pass 2, called cp2, is the final pass of the C compiler. cp2 contains the actual code generation portion of the compiler. cp2 is executed using a command line of the form

```
cp2 intermediate2 Z80 options
```

where **intermediate2** is the name of the intermediate file from cp1, **Z80** is the name of the output file containing the assembler code to be input to the Cromemco Macro Assembler, and **options** is an optional string of compiler options. The default extension of the Z80 file is **.Z80.**

The list of options follows.

option	action
-g	causes cp2 to include in the Z80 output file a directive to the asmb program to include macro expansions in the assembly listing. Use of this option is equivalent to inserting **#control macro** as the first line in the C source file. -g is overridden by **#control nmacro.**
-s	causes cp2 to include C source in the Z80 output file. The source is available only when the -s option was used for cp0. The -s option will always cause cp2 to include line numbers corresponding to the C source lines in the Z80 file, whether or not -s was used in cp0. Use of this option is equivalent to inserting **#control source** as the first line in the C program, and it may be overridden by **#control nsource.**

9.4 The Macro Assembler

The output from cp2 is input to the Cromemco Macro Assembler. This paragraph will describe the command line which will cause the assembler to assemble the input file from cp2 into a relocatable object file. A detailed description of the assembler is available in the Cromemco Macro Assembler Manual, part number 023-0039, and the manual addendum, part number 023-4001. The assembler program supplied with the C package is named Asmb.com and is the same as the assembler program supplied in the model FDA software package.

The assembler command line has the form

```
asmb Z80source.@@Z
```

where **Z80source** is the name of the file output from cp2 and @@Z instructs Asmb to find the source file in the current directory, write the relocatable object file into the current directory, and produce no assembly listing. Using @ instead of Z will cause Asmb to direct the listing to a disk file with the name **Z80source**.prn.

9.5 Link

Link is the program which loads the relocatable file output from the Macro Assembler into memory and converts it to an executable program. Link does this by converting all relative addresses into absolute addresses and searching the C library for the definitions of all functions used in the program but not defined in the program. Link searches the Cromix C function library automatically before it creates the executable program on disk.

There are two library files supplied with Cromemco C: **clib.rel** and **cdosclib.rel.** **clib.rel** is the library to use when linking a program to run under the Cromix operating system--the functions in this library use Cromix system calls to perform their tasks. **cdosclib.rel** is the library to use when linking a program to run under CDOS or the CDOS simulator--the functions in this library use CDOS system calls.

The following command line is sufficient to link a C program named **cprog** to execute under the Cromix operating system:

```
link cprog,cprog.bin/n/e
```

cprog must contain definitions for all non-library functions which it references. Link will load **cprog.rel,** automatically search **clib.rel** and load all necessary functions, and write a new file named **cprog.bin.** The **.bin** extension is a signal to the Cromix operating system that the program can execute directly under the Cromix operating system without the CDOS simulator.

The command line to link **cprog** and make a program which will execute under CDOS is:

```
link cprog,cdosclib/s,cprog/n/e
```

Link will name the executable program **cprog.com.** Should **cprog.com** ever be run under the Cromix operating system, the **.com** extension indicates that the program requires the CDOS simulator to execute CDOS system calls.

Note that Link automatically searches the file **clib.rel,** but must be explicitly instructed to search the file **cdosclib.rel.** The user must be sure to define all of his external names, such as functions and external variables, when writing a program to be linked to execute under CDOS. The Link program searches the default library only when names remain undefined at the time the **/e** Link command is entered. The default library name requested by the C compiler is **clib.rel,** and Link will search this library even after it has performed the explicitly commanded search of **cdosclib.rel** if there are any undefined names.

Now suppose that the C program named **cprog** calls a function previously compiled and assembled into a relocatable file named **fn.rel.** The command line to link the program and function into an executable program which will execute directly under the Cromix operating system would then be:

```
     link cprog,fn,cprog.bin/n/e
```

Link will load **cprog.rel, fn.rel,** search **clib.rel** for standard functions, and finally make a new file named **cprog.bin.**

Link does not recognize Cromix file pathnames. File names are limited to:

```
     eightchars.threechars
```

The above discussion should enable most users to link programs for execution. Link is fully documented in the Cromemco Link and Lib Reference Manual, part number 023-4026.

9.6 Lib

Lib is a program which is used to concatenate relocatable files into one library which may then be searched by Link in the same way that **clib.rel** is searched. The following command line will form a library file named **usrlib.rel** from the three relocatable files **fnl.rel, fn2.rel,** and **fn3.rel:**

```
     lib usrlib=fnl,fn2,fn3/e
```

Lib has many other capabilities, all of which are documented in the Link and Lib manual mentioned in the previous section.

9.7 Compiler Command File

The following Cromix command file, named **cc.cmd,** has been included with the C package for convenience.

```
     cp0 #l.c #l.p01 #2
     cpl #l.p01 #l.pl2
     cp2 #l.pl2 #l #2
     asmb #l.@@z
     link #1,#1.bin/n/e
     del #1.p01 #1.p12 cccccc.zs* #1.z80
```

It is used by typing its name followed by the name of the C source program and, optionally, a list of options for cp0 and cp2:

```
     cc cprog -s
```

To use this command file to compile and link a program to execute under CDOS or the CDOS simulator, use the Screen program to change the Link line to

```
     link #1,cdosclib/s,#1/n/e
```

9.8 Example Compilation

The following commands will compile a C source program with the name **test.c,** assemble the compiler's output to produce a relocatable file named **test.rel,** and link the relocatable file to produce an executable program named **test.bin** which will execute under the Cromix operating system.

```
     cp0 test.c test.p01
     cp1 test.p01 test.p12
     cp2 test.p12 test
     asmb test.@@z
     link test,test.bin/n/e
```

9.9 Clist

Clist is a program which reads one or more source files and displays each source line on the terminal preceded by a line number, starting at line 1 for each file. These line numbers correspond to the line numbers displayed in syntax messages from the passes of the C compiler.

To use **clist,** type:

```
     clist f1 [ f2 ...  ]
```

f2 ... are optional.

Clist can be aborted at any time by pressing CNTRL-C. Clist runs only on a Cromix system.

Chapter 10

ERROR MESSAGES

This chapter lists error messages which are displayed by the various passes of the compiler and by the runtime library routines. A pass will abort execution after displaying an error message marked as "fatal".

10.1 cp0 Error Messages

#endif without #if

There is no preceding **#if** for this **#endif.**

#if implemented as #ifdef

#ifs nested too deep

Fatal.
#ifs can be nested to a maximum of 10 levels.

#include filename not enclosed in quotes or brackets

Fatal.
A **#include** file name must be enclosed in matching quotes, **"..",** or matching angle brackets, <..>.

#include nested too deep

Fatal.
#include files can be nested to a maximum of 10 files.

ABORT: two file names required

Fatal.
cp0 must be executed with a name for the input file and a name for the intermediate file which is input to cp1.

bad digit in floating point number	This message usually means that there is a hexadecimal digit in what should be a **float** constant.
bad punctuation	The) is missing from a macro definition, or quote marks, ", do not occur where expected by cp0.
cannot create file	Fatal. cp0 cannot create the output file for some reason. This usually means that there is no available disk space.
compiler error	Fatal. cp0 found itself in an irrecoverable state, possibly caused by a logical error in the input file. One can try to isolate the error by compiling increasingly smaller parts of the program until cp0 can finish normally.
could not open file	Fatal. cp0 could not open the input file for some reason. This usually means that the file does not exist in the current directory.
extraneous token in #-line	The compiler control line contains extraneous characters on the line following all required tokens.
identifier needed	There is no identifier in the **#ifdef** line.
illegal file name	Fatal. Either the input file name or the output file name starts with a character which has an ASCII value less than the space character.

illegal preprocessor keyword	Fatal. cp0 does not recognize the word following the #.
invalid operator	The token is not an operator defined in the C language.
macro needs arguments	cp0 attempted to expand a macro **#defined** to have arguments, and there are no arguments given for this expansion of the macro.
macro table overflow	Fatal. There is a fixed amount of space allocated for the table of macro names, and it has been exhausted. One can either separate the program into several modules, or reduce the length of all macro names until cp0 can finish normally.
missing string delimiter	cp0 cannot find the right bracket, >, in a **#include** line, or it cannot find the right quote mark, ", in a string constant.
not hexadecimal digit	A hexadecimal digit is required in this context.
quotes don't balance, or constant too long	A constant can be only as long as a line and all its continuation lines, for a maximum of 512 characters.

WARNING: invalid command line argument

The only valid command line arguments are -i, -l, -p, and -s.

wrong number of macro arguments

This use of the macro does not include the number of arguments **#defined** for the macro.

10.2 cpl Error Messages

argument/declarator list mismatch	The declaration list of the function body does not match the argument list within the parentheses of the argument list. Either there is a declaration for a variable which is not in the argument list, there is no declaration for a variable in the argument list, or there is a duplicate name in the argument list.
case table full	There is a maximum of 32 cases per **switch** statement. Nested **switch** statements may each have up to 32 cases.
compiler error	cpl found itself in an irrecoverable state, possibly caused by a logical error in the source program. One can try to isolate the error by compiling increasingly smaller parts of the program until cpl can finish normally.
empty template	The tag, structure, or union being declared or defined has no declaration list enclosed within {..}, or cpl discarded incorrect declarations within the declaration list resulting in an empty list.
error from pass 0	Fatal. cpl detected an logical error in the intermediate file from cp0. This is usually caused by a flawed intermediate file, which can be remedied by running cp0 again.
expression error	This is a catch-all error message which usually accompanies a more explicit error.
function required	cpl found a left parenthesis, and the identifier to the left is not a

function name.

identifier required

A variable definition, such as **int** i[5], requires the name of a variable.

initialization error

The list of initializers is wrong in some way. There may be too many, too few, or there might be some of the wrong type.

invalid case or default

cpl detected a **case** or **default** which is not in a **switch** statement.

invalid cast

Casts can be done only to simple types, such as **int.** One cannot cast an expression to a pointer, structure, union, or type which has been **typedef**ed.

invalid constant expression

All operands in the expression must be constants.

invalid continue or break

continue and **break** are valid only within a **do, for, switch,** or **while** statement.

invalid float or double operation

Operations such as <<, >>, &, (logical and), |, %, and so on are not allowed on **floats** or **doubles.**

invalid indirection

cpl detected an attempt to apply the

	indirection operator, *, to a non-pointer identifier.
invalid initialization	
invalid pointer expression	One cannot add, multiply, or divide two pointers, shift a pointer, or add a **float** or a **double** to a pointer.
invalid reference	
invalid storage class	This storage class cannot be used at this point in the program. For example, it is incorrect to declare an **auto** variable outside of any function.
invalid storage type	The valid types are **char, int, short, unsigned, long, float,** and **double.**
invalid subscript	The array element reference contains more subscripts than given in the array definition.
lvalue required	C requires an lvalue (expression referring to an object) as an operand at this place in the source. This can happen when using =, * (indirection), or & (address operator).
missing (	cpl expected to find a (at this place in the source. This can happen in an **if, switch, for,** or **while** statement, or in a function call.

missing)	cpl expected to find a) at this place in the source. Again, this can happen in an **if, switch, for,** or **while** statement, or in a function call.
missing , or ;	cpl expected to find a **,** or **;** at this place in the source. This can happen in a **for** statement and at the logical end of an operation or statement.
missing :	cpl failed to find a : in a **switch** or a conditional expression.
missing ;	cpl expected to find a **;** at this place in the source. This can happen at the logical end of an operation or statement.
missing [	cpl expected to find a [at this place in the source. This can happen during an array definition or in an array element reference.
missing {	cpl expected to find a { at this place in the source. This can happen during a function definition or in a structure or union declaration.
missing } or ,	cpl expected to find a } or a **,** at this place in the source.
missing or invalid expression	There is something wrong with the source expression. cpl usually discards an invalid expression and continues the compilation.
missing while	cpl expected to find a **while** clause in the **do** statement.

open fail - infile	Fatal. cp1 could not open the intermediate file from cp0 for some reason.
open fail - outfile	Fatal. cp1 could not open the output file for some reason.
pointer/array required	The object preceding a [is not an array or a pointer.
read error	Fatal. cp1 got a read error while attempting to read an input file.
redefined label	There is more than one definition for the label. cp1 retains the second definition for the scope of the enclosing function.
redefined statement label	There is more than one definition for the statement label. cp1 retains the second definition for the scope of the enclosing function.
structure/union required	The object to the right of the -> operator or to the left of the . operator is not a structure or union.
tag not of same type	A structure tag is being used to define a union, or a union tag is being used to define a structure.

tag not template	The tag being used to define a structure or union has not been previously declared, or the tag declaration is not followed by {..}.
too many initializers	There are more initializers than members of the aggregate being initialized.
undefined identifier	This identifier is not defined in the program.
undefined statement label	There is no definition for the statement label.
unexpected eof	cpl found end of file while it was parsing a statement or expression. This usually means that there are mismatched braces in the source.
variable name table full	Fatal. cpl has a maximum of 4K bytes allocated for variable name storage, and this has been exhausted. cpl never deletes the names of **extern** variables from the table, but it does delete the names of **auto** variables once it has completed parsing the enclosing function. The remedy for a full table is to redefine **extern** variables as **auto** variables local to a function.
write error	Fatal. cpl got a write error while attempting to write an output file. This can happen when disk space becomes exhausted.

10.3 cp2 Error Messages

This section lists error messages which can be reported by cp2. There are three groups of error messages: those advising of a user or operating system error, non-fatal errors, and fatal errors. cp2 will continue after reporting a non-fatal error, and it will abort after reporting a fatal error.

If cp0 and cp1 have not reported any errors, cp2 should not report any errors either. Should cp2 report any non-fatal or fatal errors after both cp0 and cp1 have found no errors in the program, attempt to recompile the program starting with cp0. If cp2 reports an error again, please document the program and error message and contact Cromemco.

When cp0 or cp1 reports any errors and cp2 is executed anyway, as it might be when using a command file to compile a program, cp2 will probably report some errors. Once the errors detected by cp0 and cp1 are corrected, cp2 should report no errors.

10.3.1 cp2 User or System Errors

can't open '.z80' program output file	cp2 cannot open the output file for some reason, possibly because there is no available disk space.
can't open '.zss' program output file	cp2 cannot open a temporary output file, again perhaps caused by the lack of disk space.
can't open '.zst' program output file	cp2 cannot open the second temporary output file.
could not reopen zst or zss file	cp2 cannot reopen one of the recently created temporary files.

invalid command parameter

The allowable command line options are **-g** and **-s.**

two filenames needed

The cp2 command line must contain two names--the name of the intermediate file from cp1 and the name of the z80 output file.

10.3.2 cp2 Non-fatal Errors

This section lists the non-fatal errors which might be reported by cp2. cp2 will continue after reporting a non-fatal error. cp2 should not report a non-fatal error if both cp0 and cp1 have reported 0 errors.

attempt to take addr of constant

bad p2xx control parm

bad size to set zeroes

Boolean, 2nd arg wrong size

can't ++ or -- a constant

extending unkown vtype

float/double shift count

illegal store destination

illegal store destination -- EOBJ/LPOP

long constant not numeric

must store a constant with scon

non-numeric float/long constant

not EOBJ/LPOP/LOGICAL as 2nd boolean arg

pointer size is zero -- scaling

pointer size of zero -- unscale

stack level -- XEOE

stack level -- XRTN

unary op -- V2 not EDUM or ENULL

undefined 'where' encountered

10.3.3 cp2 Fatal Errors

This section lists the fatal errors which might be reported by cp2. cp2 will abort after reporting a fatal error. cp2 should not report a fatal error if both cp0 and cp1 have reported 0 errors.

bad 'spcl' field in op table

bad place to store a constant

bad place to store a string

bad rsize in op table

code not implemented yet

comparison op followed by invalid XEOE

inconsistent unary op

invalid string store length

invalid T type

op not handled

operator table error -- NUM

trying to zero memory with zero length

unexpected end of file

10.4 Runtime Error Messages

This section lists error messages reported by functions in the C runtime library. All runtime errors are fatal. File buffers of open files are not flushed when the program aborts.

ABORT: fatal file close error

fclose was unable to close a file opened for buffered I/O with the "e" option set in **fopen.**

ABORT: fatal file seek error

fseek was unable to set the file pointer for a file opened for buffered I/O with the "e" option set in **fopen.**

ABORT: fatal read error

fillbuff was unable to read bytes from a file opened for buffered I/O with the "e" option set in **fopen. fillbuff** is called from **getc,** which is called from **fgets.**

ABORT: fatal write error

flushbuff was unable to write to a file opened for buffered I/O with the "e" option set in **fopen. flushbuff** is called from **putc,** which is called from **fputs, printf,** and **fprintf.**

ABORT: file open error

fopen was unable to open a file for buffered I/O with the "e" option set.

ABORT: file pointer required

The program has passed a file number to an I/O function which requires a file pointer. See Chapter 4, Input and Output, for the list of functions which require file pointers.

ABORT: program/stack too big

A very large program, or a very large stack size redefinition, has caused the initial top of stack to be located in memory reserved for the operating system. Either split the program into two smaller independent programs or, in the second case, redefine the stack size so that the top of stack is not located in reserved memory.

ABORT: stack/auto space collision

The CPU stack has overflowed its limits. The stack size can be changed from the default 2K by defining an external variable named **_stack** with an intialized value which is the desired stack size. See Section 8.4, Memory Usage, for more details.